Universidad Carlos III de Madrid

DEPARTAMENTO DE ELECTRÓNICA

Grado en Ingeniería en Tecnologías Industriales

Trabajo de fin de grado

ACELERACIÓN CON GPU DE ALGORITMOS DE MATCHING PARA RECONOCIMIENTO BIOMÉTRICO MEDIANTE PATRÓN DE IRIS

Autor: Alejandro Martín-Serrano Martín

Tutor: Luis Mengibar Pozo

Leganés, Septiembre de 2014

Agradecimientos

Quiero agradecer en primer lugar a mis padres que me han apoyado con entusiasmo en todo lo que he hecho, siendo una fuente de motivación constante durante la realización del Trabajo de Fin de Grado y durante toda la carrera. Gracias por brindarme el cariño y la alegría necesaria para seguir adelante.

A mi hermana Irene, que siempre ha estado a mi lado haciéndome la vida más fácil y divertida. A mi primo Álvaro, sin duda un ejemplo como persona. A Amira por animarme a subir el último escalón de esta etapa de mi vida. A mis compañeros y también amigos de la carrera de los que he aprendido mucho y con los que he compartido momentos de estudio.

También quiero agradecer especialmente su empeño y dedicación a mi tutor Luis que me ha guiado en la realización de este proyecto.

Resumen

El siguiente trabajo consiste en la implementación del algoritmo de *matching* para reconocimiento biométrico mediante patrón de iris. El algoritmo de *matching* permite la comparación de patrones de iris para la identificación o verificación de individuos.

Por lo general, las técnicas de reconocimiento biométrico requieren de un elevado coste computacional debido a que se procesan elevadas cantidades de datos. En este trabajo se acelera el mencionado algoritmo mediante GPU (*Graphics Processing Unit*) aprovechando la potencia de estas presentan. Las tarjetas gráficas de *NVIDIA* permiten la programación de propósito general (GPGPU) para desarrollar aplicaciones que usualmente se procesan en la CPU.

La aceleración mediante GPU consiste en la paralelización del algoritmo realizándose así varias tareas simultáneamente con el objetivo de aumentar notablemente el rendimiento de la aplicación. Con el objetivo de estudiar el rendimiento de la aplicación acelerada sobre la GPU respecto a la secuencial sobre la CPU, se implementa el algoritmo de las dos maneras exponiéndose una comparativa entre ambos.

Los lenguajes de programación empleados en el desarrollo del trabajo son C para la implementación sobre la CPU y C con extensión CUDA para la implementación paralela sobre la GPU.

Para poder entender los conceptos del proyecto, se realiza una introducción teórica sobre la biometría y el reconocimiento de iris destacando el algoritmo de *matching*, así como una introducción sobre la computación paralela centrándose en la programación sobre las GPUs.

Palabras clave: Biometría, CUDA, matching, distancia de Hamming.

Abstract

The following work consists of implementing the *matching* algorithm for biometrics recognition through the iris pattern. The *matching* algorithm allows the comparisons between iris patterns for individuals identification or verification.

Generally, biometrics recognition techniques require a high computing cost due to the high amount of data being processed. In this work the mentioned algorithm is sped up through the GPU (*Graphics Processing Unit*) using their power. Video cards by *NVIDIA* allow the general purpose computing (GPGPU) to develop applications that are usually processed in the CPU.

The acceleration through GPU lies in parallelizing the algorithm processing several tasks simultaneously. The aim of this is to increase the application performance significantly. In order to study the improvement in the performance of the sped up application on the GPU with regard to the sequential application on the CPU, the algorithm will be implemented in both ways exposing a comparison between them.

The programming languages used in the development of the work are C for the implementation on the CPU and CUDA for the parallel implementation on the GPU.

In order to be able to understand the concepts of this work, it's required a theoretical introduction about biometrics and iris recognition standing out the *matching* algorithm, as well as an introduction about parallel computing focusing in programming on GPUs.

Keywords: Biometrics, CUDA, matching, Hamming distance.

ÍNDICE DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

1.1. Motivación

La aceleración de algoritmos mediante GPU (*Graphics Processing Unit*) es una forma de computación paralela que consiste en la división de un problema en tareas que se procesan simultáneamente en la unidad de procesamiento gráfico con el objetivo de aumentar el rendimiento de cómputo, permitiendo así reducir considerablemente el tiempo de ejecución requerido. Lo cual resulta muy útil aplicado a problemas que no se pueden resolver en un tiempo razonable ya que requieren un elevado número de cálculos. Esto es posible debido a que la GPU a diferencia de la CPU consta de cientos o miles de núcleos capaces de trabajar de manera simultánea.

En el presente trabajo se estudia la aceleración a través de la paralelización de algoritmos en GPU, utilizando el modelo de programación CUDA de *Nvidia*. Para ello, se ha implementado el algoritmo *matching* de reconocimiento biométrico mediante patrón de iris, en el cual se trabaja con grandes cantidades de datos que requieren muchos cálculos, lo que supone un coste computacional elevado.

El algoritmo de *matching* consiste en la comparación de patrones en forma de códigos binarios implicados en el reconocimiento de iris ocular. Para la comparación de dichos códigos se ha utilizado la distancia de *Hamming*.

1.2. Objetivos

El objetivo principal de este proyecto es la aceleración del algoritmo en GPU y el estudio del rendimiento comparándolo con su implementación secuencial en CPU.

Para alcanzar el objetivo principal es necesario cumplir los siguientes objetivos intermedios:

- Aprendizaje de programación en CUDA, incluyendo el hardware de la GPU y las herramientas necesarias de compilación para poder programar en este lenguaje.

- Conocer y entender el algoritmo de reconocimiento mediante patrón de iris, con especial atención al algoritmo de *matching* para su implementación en CUDA y para una correcta interpretación de los resultados obtenidos.

- Obtención y procesamiento de la base de datos. Se dispone de una base de datos con 756 imágenes de ojos en formato .bmp pertenecientes a 108 individuos distintos, de manera que a cada individuo le corresponden 7 imágenes distintas. Esta base de datos será procesada en *Matlab*, de donde se obtendrán las plantillas necesarias para el algoritmo de *mathing*.

- Adquirir los conocimientos necesarios en *Matlab* para la extracción de los datos de partida para la implementación del algoritmo de *matching*.

1.3. Descripción del documento

El presente documento de divide en ocho partes diferenciadas: introducción y objetivos (el presente apartado), estado del arte, trabajos realizados, conclusiones, planificación y presupuesto, y trabajos futuros.

El estado del arte corresponde con el apartado 2, en él se realiza una breve introducción a las técnicas biométricas, especialmente al reconocimiento mediante iris ocular, ya que es el objetivo de nuestro estudio. Se pasa a continuación a exponer las bases teóricas de la computación paralela con un mayor énfasis en la programación en GPUs basándose en el modelo de programación CUDA. Posteriormente se describe el algoritmo utilizado para la comparación de patrones de iris.

Los trabajos realizados corresponden a los apartados 3, 4 y 5. En el apartado 3 se describe la paralelización del algoritmo de *matching* proponiéndose varias alternativas posibles, de las cuales se han escogido las que ofrecían mejores rendimientos. En el apartado 4 se describe el trabajo computacional realizado desde la extracción de los datos necesarios hasta la implementación en CUDA proponiéndose tres modelos posibles de los que se evaluarán sus rendimientos en el apartado 5. En este apartado se realiza una breve descripción del método utilizado para las mediciones de tiempo de ejecución de las aplicaciones y se exponen los resultados obtenidos experimentalmente para, posteriormente, realizar una comparación entre los rendimientos conseguidos en la implementación secuencial y la implementación en CUDA para cada modelo propuesto.

Posteriormente, en el apartado 6, se ven las conclusiones del trabajo realizado. La planificación del trabajo completo y el presupuesto económico del proyecto se describe en el apartado 7. Por último, en el apartado 8, se presentan algunas ideas de trabajos futuros.

2. ESTADO DEL ARTE

Para una mejor comprensión del presente trabajo, en este apartado se van a exponer los fundamentos teóricos relacionados con el objeto de estudio.

En primer lugar se tratará el ámbito de la biometría exponiéndose sus características y algunas de las técnicas biométricas más comunes centrándose en el reconocimiento biométrico del iris. Posteriormente se verán las distintas formas de computación paralela prestando una mayor atención en la paralelización sobre GPUs mediante CUDA, y por último se analizará el algoritmo de *matching* empleado para la comparación de patrones de iris.

2.1. Biometría

La biometría consiste en la medida de rasgos característicos de una propia persona para su posterior identificación.

Los rasgos o características utilizados para la identificación biométrica se denominan identificadores biométricos y estos pueden ser físicos o de conducta [1]. Las características físicas son estáticas, como la huella dactilar, el iris, la geometría de la mano, etc. Mientras que las características de conducta son dinámicas, como son la firma, la dinámica del teclado o la voz, siendo esta última considerada como una característica tanto física como de conducta.

No obstante, no todos los rasgos ya sean físicos o de conducta son identificadores biométricos, sino que debe reunir las siguientes características [1]:

- *Universalidad:* La característica debe ser común entre todos los individuos de la especie que se quiere identificar.

- *Singularidad*: La característica ha de ser única del individuo, es decir, no puede haber dos individuos con una característica idéntica. Esto descartaría, por ejemplo, la estatura o la masa corporal como identificadores biométricos.

- *Estabilidad*: Debe ser una característica que permanezca estable e invariable a lo largo del tiempo y en entornos diversos.

- *Cuantificación*: La característica se debe poder medir y conocer su cantidad numérica exacta.

- *Aceptabilidad*: Indica el nivel en el que las personas están dispuestas a aceptar el uso de un determinado identificador biométrico en su vida diaria.

- *Rendimiento*: La exactitud debe ser lo suficientemente elevada para que la característica sea aceptada como identificador biométrico.

- *Usurpación*: Los sistemas de reconocimiento de ese rasgo deben ser seguros para resistir ataques fraudulentos.

En la tabla 1 se recogen algunas de los identificadores biométricos más comunes comparando sus características:

	Firma	Huella dactilar	Iris	Cara	Geometría de la mano	Voz
Universalidad	Baja	Media	Alta	Alta	Media	Media
Singularidad	Baja	Alta	Alta	Baja	Media	Baja
Estabilidad	Baja	Alta	Alta	Media	Media	Baja
Cuantificación	Alta	Media	Media	Alta	Alta	Media
Aceptabilidad	Alta	Media	Baja	Alta	Media	Alta
Rendimiento	Baja	Alta	Alta	Baja	Media	Baja
Usurpación	Alta	Baja	Baja	Alta	Media	Alta

Tabla 1. Comparación de tecnologías biométricas atendiendo a sus características [3].

La identificación biométrica requiere de análisis matemáticos y estadísticos de datos sobre una característica biológica distintiva del ser humano. Su aplicación se extiende a diversos campos como la verificación de la identidad, el control de accesos o aplicaciones forenses, como se verá más adelante.

2.1.1. Técnicas de identificación biométrica

En la actualidad, existen diversos métodos de identificación biométrica, utilizándose unos u otros según la aplicación requerida. A continuación, se describen algunas de las técnicas biométricas más comunes.

Firma

Es una de las técnicas biométricas más aceptadas debido a su frecuente utilización en documentos escritos. La firma es un identificador fiable y muy aceptado por la sociedad debido a su uso tradicional y a su fácil realización [1]. Sin embargo, la

firma de un individuo varía a lo largo del tiempo y en ocasiones en cada ejecución, por lo que su identificación resulta más compleja.

Huella dactilar

La identificación de personas a través de la huella dactilar es el método más extendido ya que es un método socialmente muy aceptado y fiable.

Su gran fiabilidad se debe a que se trata de un identificador unívoco y morfológicamente invariable en el tiempo debido a que la huella dactilar crece, sin alterar su forma, en proporción al crecimiento del individuo.

Se emplea para la verificación e identificación de de personas comparando una huella digital con la huella patrón, en el caso de verificación, o con una base de datos de huellas, para la identificación.

Existen distintas tecnologías para la adquisición y digitalización de huellas dactilares sin necesidad de impregnar el dactilograma natural en tinta, como *scanners* ópticos, capacitivos o por ultrasonidos [1].

Figura 2.1. Huella dactilar [2].

Iris

El iris es uno rasgos más potentes utilizados en la biometría debido a su elevada estabilidad y sobre todo a su unicidad, siendo incluso mayor que la de la huella dactilar, y además es distintivo para cada uno de los ojos de una persona.

Figura 2.2. Captura de imagen de iris [5].

La captura de la imagen del iris se hace mediante una cámara de alta resolución y a una distancia lo suficientemente cerca para captar únicamente el ojo de la persona [1].

Facial

El rostro es uno de los identificadores biométricos más aceptados y es usado tanto para la identificación como para la autentificación de sujetos.

Se parte de la imagen del rostro a identificar y se realiza un análisis de algunas de las características físicas más unívocas y estables, como la distancia entre pupilas o la posición de la nariz [1].

Esta técnica tiene la desventaja de que no se distingue si el identificador es un rostro real o se trata de una fotografía, por lo que su uso para control de acceso no es recomendable. No obstante, también existen técnicas 3D, las cuales presentan una mayor seguridad ya que se puede obtener otro tipo de información característica como la información espacial del rostro o la textura de la piel [9].

Geometría de la mano

Esta técnica consiste en la medición de determinados rasgos del dorso y del perfil de la mano, como la altura y la anchura de los dedos, la anchura de la palma, desviaciones de distancias entre puntos característicos, etc. [1].

Para la toma de datos se debe obtener una imagen de la mano mediante una cámara digital, en donde la mano se apoya sobre una plataforma con varios topes que ayudan la colocación correcta de los dedos de la mano como se muestra en la figura 2.3.

Figura 2.3. Lector de manos [6].

La aplicación más habitual de reconocimiento de geometría de la mano es el control de accesos físicos, a menudo, acompañado de la introducción de una clave o PIN asociado a la plantilla patrón de la mano comparada.

Voz

La voz es un identificador biométrico muy aceptado ya que es el rasgo que usamos con mayor frecuencia para comunicarnos entre personas. Depende tanto de las características fisiológicas como de comportamiento del propio sujeto, si bien, estas últimas cambian con la edad y estado de ánimo de cada individuo. Para su reconocimiento se debe realizar un pre-procesado de la señal acústica obtenida y buscar las similitudes con otros patrones.

2.1.2. Estructura general de un sistema de reconocimiento biométrico

Un sistema de reconocimiento biométrico consta de varias etapas que se encuentran relacionadas entre sí, de manera que cada una de las etapas dependen de las etapas predecesoras. En la figura 2.4 se muestra el esquema del funcionamiento de un sistema de reconocimiento biométrico.

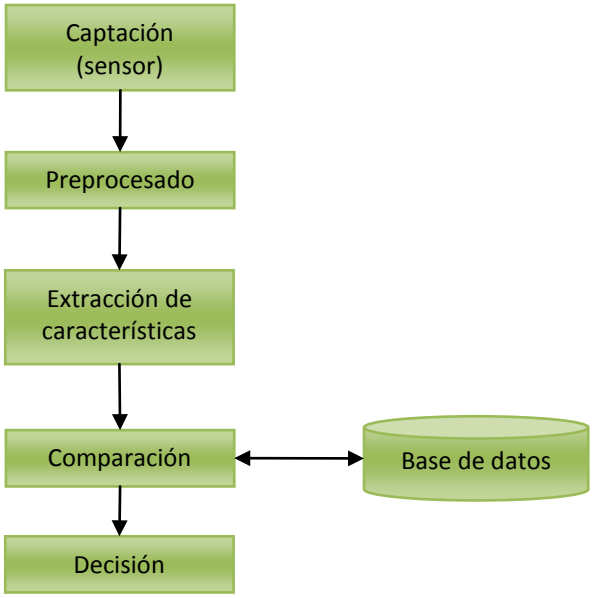

Figura 2.4. Esquema general de un sistema de reconocimiento biométrico.

A continuación se resumen las etapas de las que se compone un sistema reconocimiento biométrico como el representado en la figura anterior:

- **Captación:** se adquiere la información biométrica a través de sensores y se digitaliza para ser procesada posteriormente. Esta etapa debe estar estandarizada para realizarse siempre en condiciones de captura lo más similares posibles, como por ejemplo la distancia o la posición del sensor respecto al rasgo biométrico. Además se deben tener en cuenta los factores ambientales (nivel de iluminación) y el estado físico del dispositivo (deterioro, calibración) para adquirir información de alta calidad que determina el resultado final del reconocimiento [1].

- **Preprocesado:** a veces es necesario acondicionar la información adquirida para eliminar posibles ruidos producidos en la etapa anterior [8]. En la mayoría de los casos los datos biométricos como imágenes estáticas, vídeos o sonidos ocupan un espacio de memoria elevado, por lo que se deben comprimir. La compresión de los datos supone pérdidas de información [1].

- **Extracción de características:** en esta etapa se aísla la información necesaria para el proceso de reconocimiento, extrayendo únicamente aquellas características relevantes para su posterior comparación [8].

- **Comparación:** en esta etapa se comparan la características extraídas en la fase anterior con los patrones almacenados en la base de datos.

- **Decisión:** el proceso de verificación e identificación finaliza con la medición de un índice que se compara con un umbral que determina si la identificación o la verificación es satisfactoria [1].

2.1.3. Modos de operación

En un sistema de reconocimiento biométrico se pueden distinguir dos modos de operación, como sistema de verificación o de identificación [8]:

- **Verificación o autentificación:** se valida la identidad de un individuo comparando el rasgo biométrico con su propio patrón previamente almacenado en la base de datos. Únicamente se realiza una comparación y el sistema de verificación deberá determinar si ambos patrones se corresponden al mismo sujeto.

- **Identificación:** consiste en la búsqueda en la base de datos de patrones coincidentes con el objetivo de reconocer a un usuario. Se realizan múltiples comparaciones entre el rasgo biométrico a identificar y los patrones almacenados en la base de datos. El sistema de identificación deberá establecer la identidad del sujeto o por el contrario, indicar que no se encuentra en la base de datos.

2.1.4. Evaluación del rendimiento

Para conocer el grado de fiabilidad de un sistema de reconocimiento biométrico es necesario obtener una medida objetiva de su rendimiento, para ello se emplean tasas de acierto que miden los distintos errores que puede presentan el sistema. A continuación se describen algunos de los más significativos [1]:

- **FAR (*False Acceptance Rate*):** Probabilidad de que el sistema indique incorrectamente como coincidente una muestra de un impostor con un patrón de otro usuario.

- **FRR (*False Rejection Rate*):** Probabilidad de que el sistema indique incorrectamente como no coincidente una muestra de un usuario con un patrón del mismo usuario.

- **FER (*Failure-to-enroll Rate*):** Proporción de fracasos en obtener un patrón de una muestra. Normalmente, debido a muestras obtenidas de baja calidad.

En la figura 2.5 se puede observar que existe un solapamiento entre las curvas FAR y FRR, por lo que resultaría imposible acertar en la totalidad de los casos. El valor de estas tasas es dependiente del umbral de decisión que se establezca, variando ambos en sentido opuesto. De manera que, al aumentar el umbral se consiguen valores menores del FAR y mayores del FRR, haciendo el sistema más restrictivo y aumentando la probabilidad de dar por malas muestras buenas, mientras que si el umbral disminuye ocurrirá lo contrario, pudiendo dar por buenas muestras impostoras. El valor en el que se igualan los ratios FAR y FRR se denomina **EER (*Equal Error Rate*)**, este valor es popularmente utilizado como el umbral de decisión para cuyo valor del FAR o del FRR indica el rendimiento del sistema biométrico.

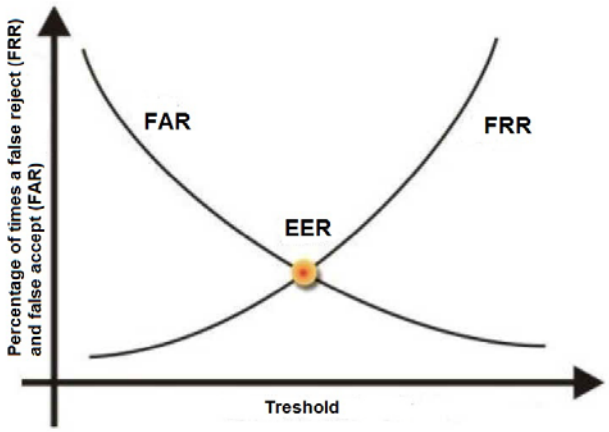

Figura 2.5. Representación curvas FAR y FRR frente al umbral [14].

2.1.5. Aplicaciones de la biometría

Con el uso de las nuevas tecnologías y el auge de Internet, son cada vez más frecuentes las consultas de nuestra cuenta corriente, las transacciones bancarias o las compras a través de Internet, es por ello por lo que la seguridad se ha convertido en una necesidad inmediata. La biometría se presenta como una herramienta fiable en los sistemas de seguridad evitando los problemas de los métodos tradicionales de identificación mediante contraseñas. Las aplicaciones biométricas pueden clasificarse de la siguiente manera [3]:

- Aplicaciones **comerciales:** Dentro del ámbito comercial, los sectores en los que la biometría cobra mayor importancia son la sanidad y el financiero. Aplicándose a la identificación inequívoca de pacientes en hospitales, control de accesos físicos en bancos, aseguradoras u otro tipo de instalaciones, uso de cajeros automáticos, transacciones bancarias, control de accesos a dispositivos electrónicos como móviles, PDAs, etc.

- Aplicaciones **gubernamentales**: En el ámbito gubernamental es sin duda donde hay mayor importancia y desarrollo de las tecnologías biométricas. Se emplea para el control de fronteras, documentos nacionales como DNI, pasaporte, carnet de conducir, tarjetas de identidad o firma electrónica.

- Aplicaciones **forenses**: El uso de la biometría en aplicaciones forenses data prácticamente desde la aparición de las tecnologías biométricas. Su aplicación se emplea para la identificación de cadáveres, determinación de parentesco, seguridad en prisiones, investigación criminal, identificación de terroristas, etc.

2.1.6. Inconvenientes y limitaciones de la biometría

A pesar de las ventajas que aporta la biometría, ésta también presenta ciertos inconvenientes y limitaciones como por ejemplo, la variación de las características biométricas a la hora de su adquisición o el rechazo social por violación a la privacidad.

2.1.3.1 Inconvenientes

La biometría presenta ciertos inconvenientes, principalmente de aceptación social y privacidad para los usuarios. Algunos de los inconvenientes más destacados son [10]:

- *Aceptación social*: La aceptación social de los sistemas biométricos está estrechamente relacionada con comodidad y facilidad con la que un usuario interacciona con el sistema. Si el sistema biométrico exige participación activa o incomodidad en la adquisición del rasgo, los usuarios podrían rechazarlo. Como contrapartida, si el sistema no requiere de la participación del usuario, un individuo podría ser identificado sin que éste se dé cuenta.

- *Privacidad*: Algunos rasgos biométricos pueden proporcionar más información de la necesaria para la identificación del mismo, un ejemplo de esto es el ADN, del cual se puede extraer información acerca de patologías médicas. Otra

amenaza a la privacidad es la identificación involuntaria de rasgos visibles en los que no sea necesaria la interacción con el usuario.

2.1.3.2 Limitaciones

Las limitaciones de la identificación biométrica se deben principalmente a la variación de los rasgos biométricos por el entorno en el que son capturados o la interacción con el usuario en la adquisición del identificador. A continuación se resumen algunos de los problemas más comunes [10]:

- *Captura imperfecta*: Las condiciones en las que se captura un identificador biométrico no son las ideales, existen numerosas variables que pueden impedir una buena captura del rasgo. Por ejemplo, la iluminación en los sistemas de captura de reconocimiento facial, mal contacto con el sensor, sensor sucio o en mal estado, orientación errónea del ojo en sistemas de reconocimiento de iris, etc.

- *Identificadores irreproducibles*: Los identificadores biométricos pueden verse alterados por lesiones, impidiendo su identificación. A veces, estos cambios pueden ser irreversibles.

- *Unicidad*: Aunque un rasgo biométrico debe ser unívoco entre individuos, existe la posibilidad de que hayan similitudes entre los rasgos de varios usuarios sin que sea posible la discriminación entre ellos.

- *No universalidad*: Es posible que algún sujeto carezca de cierto rasgo biométrico, por ejemplo, no saber escribir en reconocimiento de escritura, mudez de un sujeto en de un sistema de reconocimiento de voz, etc.

- *Robo de identidad*: Un identificador biométrico puede ser imitado usurpando la identidad de un usuario con el fin sortear el sistema de reconocimiento, como por ejemplo, la imitación de la voz o la firma.

- *Coste computacional*: Los métodos de identificación biométrica son técnicas complejas que requieren de una gran cantidad de cálculos, por lo que su ejecución en una computadora es costosa.

2.2. Reconocimiento biométrico del iris

Las características fisiológicas dotan al iris de gran estabilidad y poder discriminatorio, lo que ha potenciado que el iris sea uno de los métodos biométricos de mayor fiabilidad. Debido a ello, la utilización del iris como identificador biométrico ha crecido significativamente con la necesidad de los sistemas de reconocimiento para la verificación de identidad y con el desarrollo de las tecnologías asociadas.

En este apartado se expone la técnica de reconocimiento biométrico mediante patrón de iris. Para ello, se van a introducir algunos de los conceptos básicos sobre la anatomía del ojo humano prestando una mayor atención en el iris, para posteriormente explicar las fases que conforman el proceso de reconocimiento: captura de la imagen del iris, segmentación, normalización, codificación y *matching*. Por último se comentará la base de datos utilizada en el presente trabajo.

2.2.1. Anatomía del ojo humano

El ojo, también llamado globo ocular, es una estructura de forma aproximadamente esférica en el que su diámetro mayor es de 24 mm desde la parte anterior a la posterior [2]. La estructura de un ojo se puede ver en la figura 2.6. El globo ocular se encuentra recubierto por tres capas denominadas externa, media e interna [1].

La capa externa se compone por la *esclerótica*, que es una membrana fibrosa de color blanco, cuya parte visible se conoce comúnmente como "blanco de los ojos", y la *córnea*, que es una capa transparente que recubre el *iris* y la *pupila*.

La capa media, *úvea*, se divide en tres estructuras diferenciadas, los *coroides*, que proporciona el oxígeno y los nutrientes necesarios a la mayoría de la parte posterior del globo ocular, y se encuentra situada entre la esclerótica y la retina; el *cuerpo ciliar*, situado entre el coroides y el iris, está formado por procesos ciliares y el *músculo ciliar*, responsables de la producción del humor acuoso y de la acomodación del *cristalino;* y el *iris* (que se verá con mayor detalle posteriormente).

Por último, la capa interna denominada *retina*, que es la capa sensorial del ojo donde se transforma la luz que incide sobre ella en impulsos nerviosos que son enviados al cerebro por el nervio óptico.

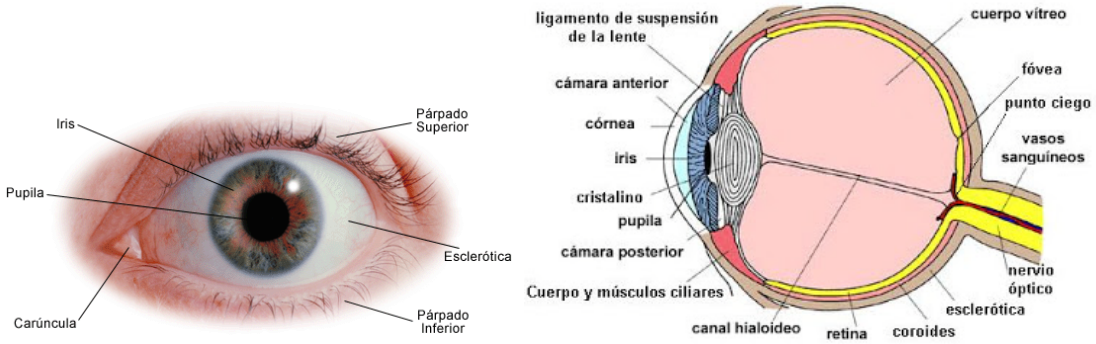

Figura 2.6. Anatomía del ojo [11] [12].

A continuación se describe con más detalle el iris, ya que es el objetivo de este trabajo.

2.2.2.1 Iris

El iris es una estructura perteneciente a la capa media del ojo, situado entre la córnea y el cristalino y posee una apertura circular en su centro denominada *pupila*, que actúa como diafragma regulando la cantidad de luz que entra en el ojo. El iris está compuesto de un *estroma* con células pigmentadas (*melanocitos*), que determinan el color del iris, y de un epitelio en el que se encuentran los músculos *esfínter* y *dilatador* [2].

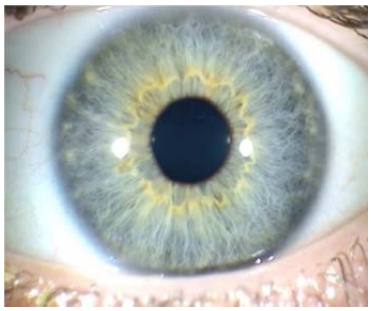

Figura 2.7. Patrón de iris [13].

Como se puede observar en la figura 2.7, el iris presenta un patrón muy complejo, que varía de una persona a otra, incluso en una misma persona del ojo izquierdo al derecho. Además, el patrón del iris permanece invariante durante prácticamente toda la vida de un individuo. A continuación se mencionan las

cualidades que permiten que el iris sea utilizado para aplicaciones de identificación biométrica [1]:

- Es visible desde el exterior debido a la transparencia de la córnea, lo que permite que el iris sea capturado con una fotografía.

- Elevada unicidad. El patrón del iris contiene más información que permite identificar a una persona unívocamente que el patrón de una huella dactilar.

- Estabilidad a lo largo del tiempo y ante cambios en el entorno.

- La falsificación del iris de una persona sólo se puede hacer mediante operaciones quirúrgicas que podrían dañar la visión.

- El hecho de que el tamaño de la pupila varíe en respuesta a cambios de iluminación o con iluminación fija, presenta una ventaja para la detección de vida de un sujeto en los sistemas de reconocimiento biométrico.

2.2.2. Captura de la imagen del iris

La captura de la imagen del iris es el primer paso en el proceso de reconocimiento del mismo. Existen dos alternativas para su captura, que son, el uso de cámara fotográfica digital o el uso de cámaras de video. El uso de cámaras de alta resolución es fundamental, ya que el rendimiento del sistema de reconocimiento depende en gran medida de la calidad de la imagen para poder analizar la textura del iris.

Para conseguir el nivel de detalle suficiente, la imagen debe capturase a una distancia cercana que permita obtener únicamente la imagen del ojo con alta resolución y sin que el sujeto se vea amenazado por la cámara. Esto supone un inconveniente ya que al tratarse de una técnica que requiere de la participación del usuario, el nivel de aceptación de este tipo de reconocimiento es menor.

Otro factor importante y problemático es la iluminación del entorno en el que se realiza la captura. El entorno de captura debe tener una iluminación adecuada, no molesta para el usuario, que permita obtener una imagen de alta calidad y además, evitando la reflexión de la luz sobre la córnea del ojo. Para ello se puede emplear luz infrarroja, con la que se puede aportar la iluminación necesaria sin molestar al usuario.

Cuando se obtiene la imagen del ojo, lo habitual es que ésta no contenga el iris en su totalidad, sino que parte del iris esté oculto por los párpados y las pestañas, tal y como se muestra en la figura 2.8.

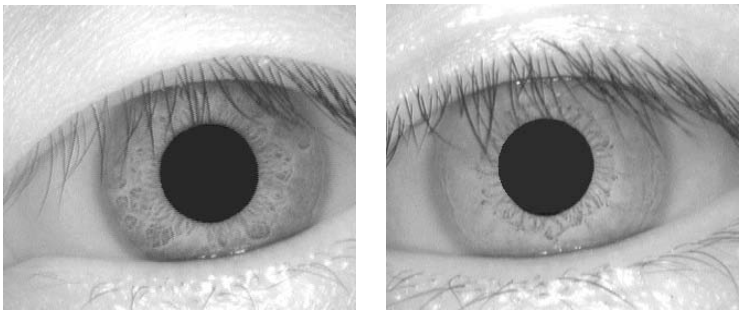

Figura 2.8. Captura de la imagen de iris. Imágenes pertenecientes a la base de datos CASIA.

2.2.3. Segmentación

Una vez capturada la imagen del iris, es necesario localizar el iris en la imagen para extraer su textura y excluir el ruido originado por los párpados, las pestañas o los posibles reflejos existentes. Antes de procesar la imagen, si ésta es en color, hay que convertirla a blanco y negro.

El iris puede ser aproximado por dos circunferencias casi concéntricas, correspondientes a los bordes exterior e interior del iris. Por lo que, para su localización es necesario detectar ambas circunferencias en la imagen. Existen varios métodos para realizar esto, de entre los que destacan *la transformada circular de Hough* y *el operador integro-diferencial de Daugman*, pudiéndose ampliar la información en [2] [15].

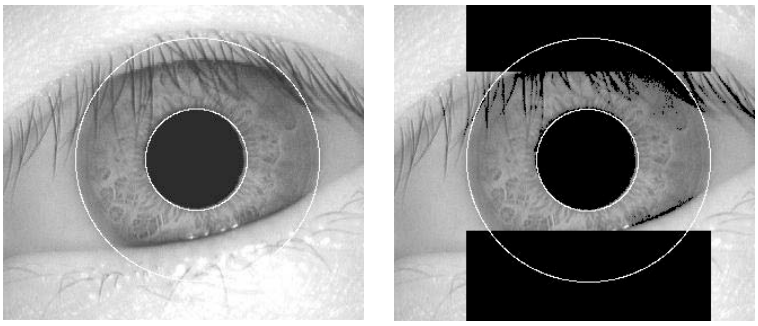

Figura 2.9. Segmentación del iris y detección de párpados y pestañas (Imágenes de la base de datos CASIA-IrisV1 procesadas en Matlab).

2.2.4. Normalización

La normalización consiste en fijar las dimensiones que pueden variar en la etapa de captura de la imagen para permitir la comparación posteriormente. Los cambios producidos entre las imágenes adquiridas son debidas principalmente a las variaciones en el tamaño del iris a causa de la dilatación de la pupila ante distintos niveles de iluminación. Otras de las razones por las que esto ocurre se deben a la distancia en la adquisición de la imagen, la rotación de la cámara, la inclinación de la cabeza y la rotación del ojo.

Una de las técnicas más empleadas para la normalización del iris es la conocida como *Daugman's Rubber Sheet Model* [15], que consiste en crear una imagen rectangular con dimensiones fijas. De manera que, cada punto de la imagen del iris se referencia con un par de coordenadas polares $r \in [0,1]$ y $\theta \in [0, 2\pi]$.

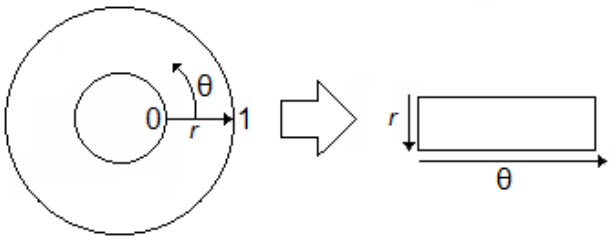

Figura 2.10. Daugman's Rubber Sheet Model [15].

Este modelo permite que todos los iris sean representados con dimensiones fijas, independientemente del tamaño de la imagen o de la dilatación de la pupila. Sin embargo, este modelo no corrige las desviaciones debidas la rotación. Por lo que, este proceso se llevará a cabo en el proceso de *matching*.

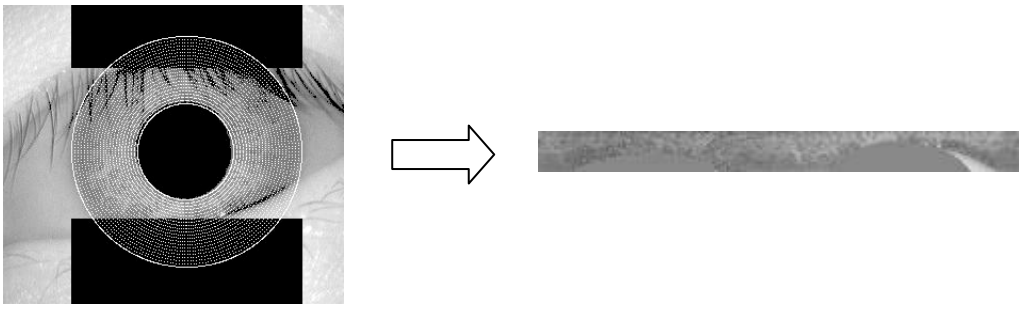

Figura 2.11. Normalización del iris (Imágenes de la base de datos CASIA-IrisV1 procesadas en Matlab)..

2.2.5. Codificación

Una vez realizadas las etapas de segmentación y normalización, se implementa el proceso de codificación en donde se extraen las características de la textura del iris, obteniéndose el código del iris (*iriscode*). Únicamente deben extraerse aquellas características que permitan realizar una comparación entre plantillas. Habitualmente, para la extracción de las características se utilizan métodos basados en filtros aplicados a la imagen de entrada, como los *filtros de Gabor* (propuesta por J.Daugman) o los *filtros de Log-Gabor*, u otros métodos como la codificación *Wavelets*, pudiéndose encontrar más información en [1].

2.2.6. Matching

El proceso de *matching* consiste la comparación de dos patrones una vez obtenidos los códigos del iris, y decidir si corresponden al mismo ojo. Existen varios algoritmos de *matching*, sin embargo, el más utilizado es el basado en la *distancia de Hamming* debido a su alto porcentaje de acierto y a su simplicidad. La distancia de *Hamming* cuenta el número de bits que difieren entre dos plantillas binarias, las cuales tienen asociadas unas máscaras de ruido de las regiones que pertenecen a la textura del iris, como los párpados y las pestañas.

El algoritmo de *matching* también debe tener en cuenta las desalineaciones producidas por la rotación de la imagen que no se han corregido en el proceso de codificación.

En el apartado 2.5 se describe, con mayor detalle, el algoritmo de *matching* basado en la distancia de *Hamming*, que ha sido el método utilizado en la realización del presente trabajo.

2.2.7. Base de datos *CASIA*

En la realización del presente trabajo se ha utilizado la base de datos de imágenes de iris *CASIA-IrisV1*, proporcionado por la *Chinese Academy of Sciences Institute of Automation* (*CASIA*). Se ha escogido esta base de datos porque es la empleada en el sistema de reconocimiento de iris implementado en *Matlab*, que es el software utilizado para ejecutar el programa hasta la etapa de codificación. Etapa de la que se extraen los datos del *iriscode* que se procesarán en el algoritmo de *matching*.

Ante la escasez de datos para las investigaciones de reconocimiento de iris se decidió crear una base de datos que *National Laboratory of Pattern Recognition*, *Institute of Automation* y *Chinese Academy of Science* ponen a disposición de forma gratuita [16]. El sistema CASIA utiliza ocho iluminadores infrarrojos colocados alrededor del sensor para asegurarse una iluminación uniforme del iris, ubicando los reflejos de los iluminadores en la pupila. Una vez capturadas todas las imágenes, se detectaron las regiones pertenecientes a la pupila y se sustituyeron por regiones circulares de intensidad constante con el objetivo de ocultar los reflejos debidos a los iluminadores, lo que hace que sea más fácil detectar la frontera entre la pupila y el iris para la extracción de las características del iris. Las imágenes fueron capturadas con un dispositivo desarrollado por CASIA a una distancia cercana al ojo. En la figura 2.12 se puede ver el sistema de captura CASIA a la izquierda, y a la derecha se muestran los reflejos debidos a los iluminadores centrados en la pupila.

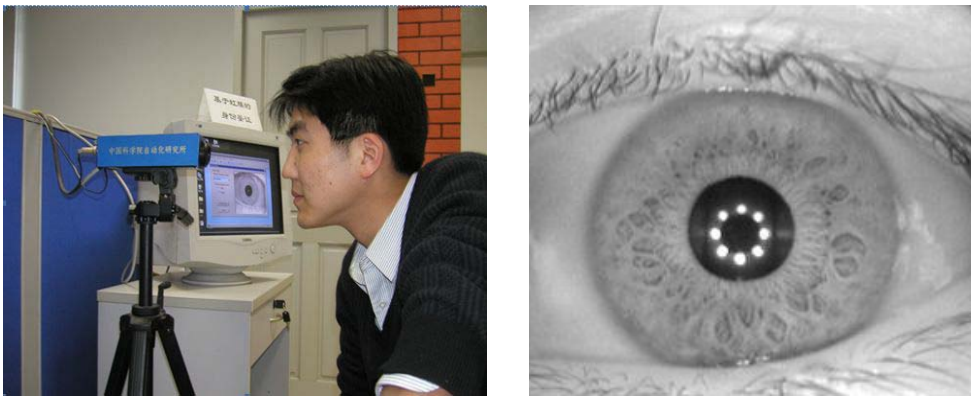

Figura 2.12. Adquisición del iris para la base de datos CASIA. Reflejo de los iluminadores centrados en la pupila [16].

En la base de datos *CASIA-IrisV1* se incluyen 756 imágenes pertenecientes a 108 ojos distintos. Para cada uno de los ojos se capturaron 7 imágenes en dos sesiones, tres en la primera y cuatro en la segunda. Las imágenes recopiladas están guardadas en formato .bmp con resolución 320x280. En la figura 2.13 se muestran dos imágenes de ojos pertenecientes a la base de datos utilizada tomadas en distintas sesiones.

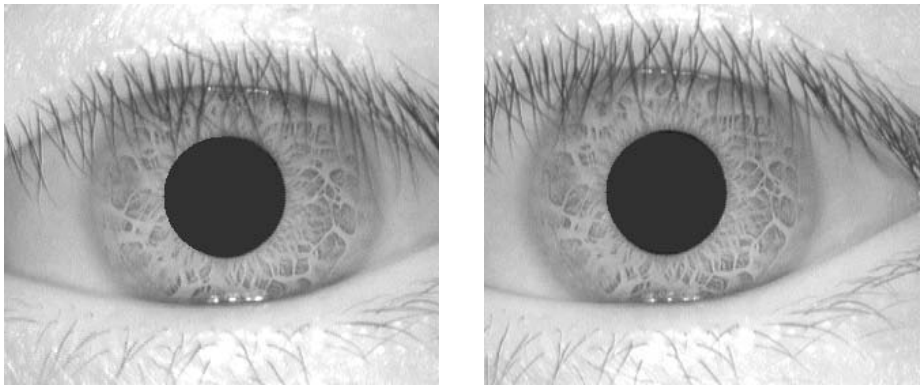

Figura 2.13. Imágenes de ojos tomadas en dos sesiones distinatas Imágenes de la base de datos CASIA-IrisV1 .

2.3. Computación paralela

La computación paralela, a diferencia de la secuencial, consiste en la ejecución simultánea de instrucciones en varios procesadores, permitiendo que muchos problemas complejos desde el punto de vista computacional puedan implementarse en un tiempo razonable. Su uso está cada vez más extendido ante la dificultad de aumentar la frecuencia de las computadoras secuenciales, ya que supone una relación coste/prestaciones muy elevada debido a que la frecuencia está muy limitada por el consumo de energía [18].

En este apartado se expondrá en primer lugar las limitaciones en el rendimiento de la computación paralela, basándose en la ley de *Amdahl*, y posteriormente las distintas formas de computación paralela así como algunas de las computadoras empleados para tal fin. Por último se expondrán algunas de las aplicaciones que presenta la computación paralela en los distintos ámbitos.

2.3.1. Ley de Amdahl

Cada algoritmo de implementación paralela presenta una parte secuencial (no paralelizable) y otra paralela. El rendimiento del algoritmo está limitado por el grado de paralelización conseguido [20].

Teóricamente, si se emplea el doble de procesadores, el tiempo de ejecución de un algoritmo debería reducirse a la mitad. Sin embargo, esto no es posible para la gran mayoría de los programas, ya que éstos no pueden ejecutarse completamente en paralelo, siempre hay parte del código del programa que debe ejecutarse en serie.

La *ley de Amdahl* establece la aceleración con la que un algoritmo es capaz de ser ejecutado, teniendo en cuenta el número de procesadores empleados y las porciones paralelas y secuenciales del programa. La ley de *Amdahl* sigue la siguiente expresión:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Donde S es la aceleración esperada, P es la fracción del programa paralelizable, N es el número de procesadores, y $(1-P)$ representa la fracción secuencial del programa.

De la anterior expresión se puede extraer la conclusión de que al aumentar el número de procesadores, aumenta la aceleración, pero existe un límite en el que la aceleración no puede aumentar más, ya que se encuentra limitada por la fracción secuencial del programa, como se muestra en la siguiente expresión:

$$\lim_{N\to\infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{1-P}$$

En la gráfica siguiente se muestra un ejemplo de la ley de Amdahl, en la que se representan las aceleraciones en función del número de procesadores empleados para varios algoritmos con distinta porción paralela:

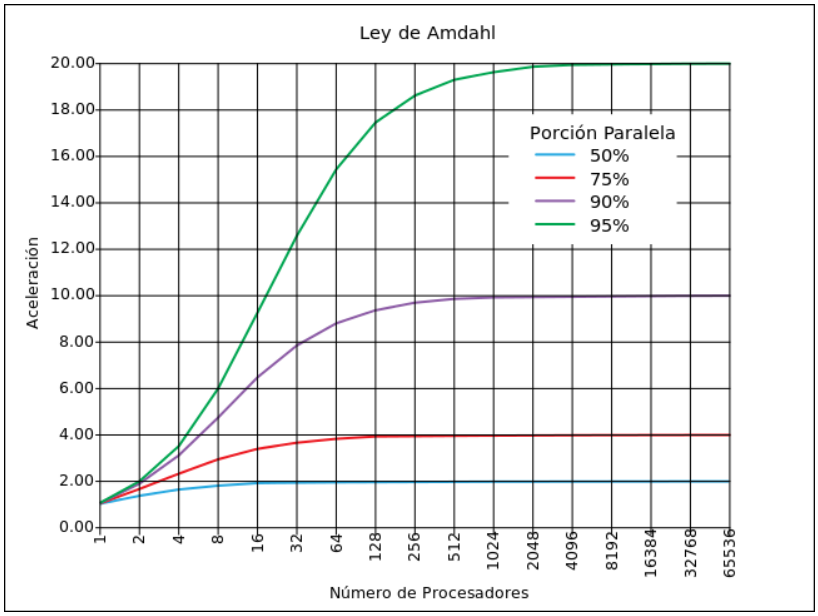

Figura 2.14. Representación gráfica de la ley de Amdahl [18].

2.3.2. Tipos de paralelismo

Las distintas formas de computación paralela se pueden clasificar de las siguiente manera[18]:

- **Paralelismo a nivel de bit:** consiste en aumentar el tamaño de la palabra que puede procesar una computadora, pudiéndose manejar mayor cantidad de información por ciclo de reloj reduciendo el número de instrucciones necesarias en las operaciones. Los microprocesadores han ido evolucionando, llegándose a aumentar el tamaño de la palabra hasta 64 bits.

- **Paralelismo a nivel de instrucción:** consiste en realizar simultáneamente el mayor número de operaciones independientes posibles sin alterar al resultado del problema.

- **Paralelismo a nivel de datos:** consiste en emplear varios procesadores de manera que cada uno de ellos realice la misma operación o secuencia de operaciones sobre un conjunto de datos. Este tipo de paralelismo se aplica normalmente a vectores y matrices en los que se deben realizar las mismas operaciones para cada uno de sus elementos.

- **Paralelismo de tareas:** consiste en realizar tareas u operaciones distintas asignándolas a diferentes procesadores, de manera que cada procesador ejecutará aquellas tareas que les ha sido asignada, a diferencia del paralelismo a nivel de datos, donde cada procesador realiza las mismas operaciones.

2.3.3. Tipos de computadoras

Las computadoras pueden clasificarse según su arquitectura y la forma en la que se estructuran sus procesadores. La taxonomía de *Flynn* [22] es una de las clasificaciones más conocidas. Ésta clasifica las computadoras en cuatro grupos basándose en el número de instrucciones y flujo de datos que pueden procesarse simultáneamente:

- **SISD** (Instrucción única, datos únicos): Las instrucciones se ejecutan secuencialmente. A esta categoría pertenecen la mayoría de las computadoras secuenciales.

- **SIMD** (Instrucción única, datos múltiples): Cada procesador ejecuta la misma instrucción sobre un conjunto de datos (empleada en paralelismo a nivel de datos).
- **MISD** (Instrucción múltiple, datos únicos): Varios procesadores ejecutan instrucciones distintas sobre el mismo flujo de datos. Esta categoría es poco habitual.
- **MIMD** (Instrucción múltiple, datos múltiples): Tienen varios procesadores independientes que trabajan sobre distintos datos. La mayoría de los multiprocesadores pertenecen a esta categoría.

2.3.4. Aplicaciones de la computación paralela

La computación paralela es empleada para la ejecución de programas de coste computacional elevado. Las aplicaciones más comunes se dan en campos de ciencia e ingeniería, y de industria y comercio. A continuación se detallan, con algunos ejemplos, cada uno de estos campos de aplicación [17] [21]:

- **Ciencia e ingeniería:** El procesamiento paralelo en este campo es empleado para resolver grandes problemas de elevada complejidad como problemas de modelización y simulación. Algunos ejemplos de aplicaciones en ciencia e ingeniería que se pueden citar son:

    - Modelización del clima, cuerpos celestes.
    - Diseño de aviones.
    - Bioinformática: análisis de secuencias genéticas.
    - Química cuántica.
    - Ciencia de los materiales.
    - Análisis numéricos.
    - Exploración sísmica.

- **Industria y comercio:** Actualmente la paralelización de estas aplicaciones se emplea para el procesamiento de elevadas cantidades de datos. Alguna de las aplicaciones son:

    - Aceleración de búsquedas en bases de datos.
    - Exploración de petróleo.
    - Modelización económica y financiera.
    - Avance medios audiovisuales y entretenimiento.

2.4. CUDA

El nombre de CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) [24]. Hace referencia tanto a la plataforma de cómputo paralelo como al modelo de programación creado por *NVIDIA*. Esta forma de computación paralela hace uso de la GPU (Unidad de Procesamiento Gráfico) aprovechando la arquitectura multinúcleo con el objetivo de aumentar el rendimiento de cómputo.

El lenguaje utilizado en la programación en CUDA es una extensión del lenguaje *C* y *C++* que permite implementar el procesamiento de tareas y datos de forma paralela. También se pueden usar otros lenguajes de alto nivel como *Fortran*, *Python* o *Java* [25].

CUDA funciona en todas las GPUs NVIDIA a partir de la serie G8x, incluyendo *GetForce*, *Quadro* y *Tesla*.

Con el transcurso del tiempo, las GPUs han ido adquiriendo un nuevo uso distinto del procesamiento gráfico, en donde se aprovecha la gran potencia de cálculo de las GPUs para aplicaciones no relacionadas con el procesamiento gráfico, dando lugar a lo que se conoce como *programación de propósito general sobre GPUs* (*GPGPU*). Y han evolucionado rápidamente en procesadores con múltiples núcleos especializados en computación masiva y altamente paralela, como se muestra en la figura 2.15.

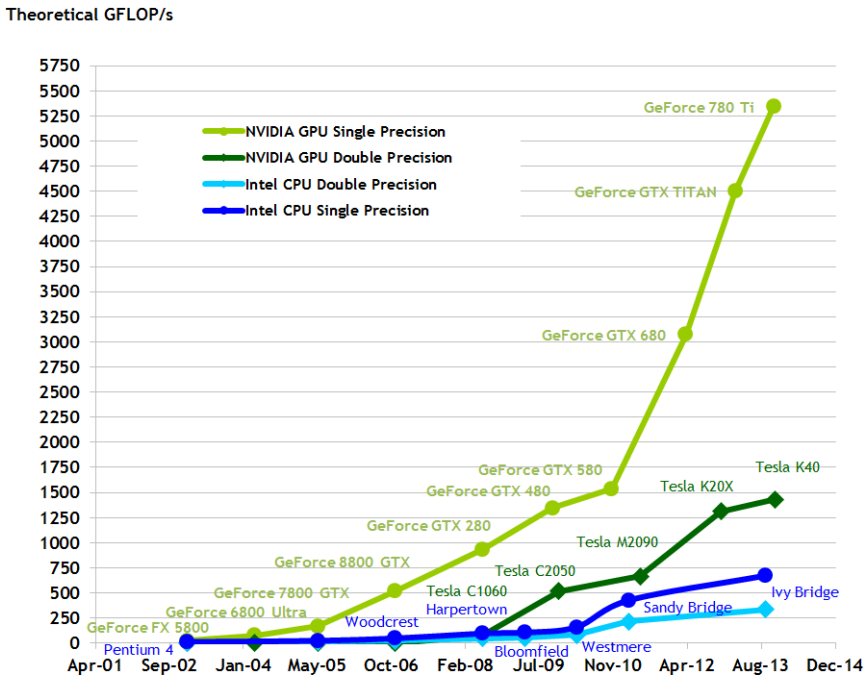

Figura 2.15. Operaciones en punto flotante por segundo para la CPU y la GPU [26].

Las tarjetas gráficas, a diferencia de las CPUs, están diseñadas de forma que la mayoría de los transistores estén dedicados al procesamiento de datos más que a la caché de datos y al control de flujo [26], como se representa en la figura 2.16. Este tipo de diseño es apropiado para problemas que usan grandes conjuntos de datos en los que se puede aplicar las mismas operaciones de forma paralela.

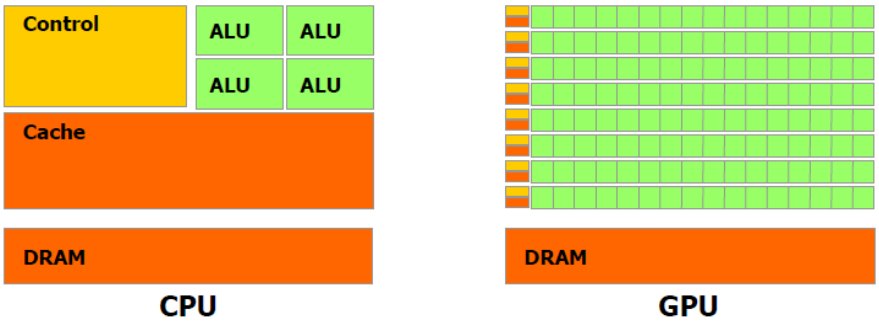

Figura 2.16. Comparación del uso de transistores en CPU y GPU [26].

2.4.1. Modelo de programación

El modelo de programación en CUDA consta de una CPU, a la que se denomina *host*, que alberga y controla el comportamiento la tarjeta gráfica, y de una GPU, denominada *device*, que se encarga del procesamiento paralelo.

El host se encarga de ejecutar el código secuencial e invoca *kernels* que son ejecutados en el device. Un *kernel* es una función, siempre de tipo *void*, que se ejecuta muchas veces en paralelo por hilos diferentes. Se define usando el atributo *__global__* en la declaración, como se muestra en el ejemplo de la figura 2.17.

```
__global__ void kernel (int *A, int *B, int *C)
```

Figura 2.17. Declaración de un kernel en CUDA

Cuando se invoca un *kernel* se especifica el número de hilos y de bloques con el que debe trabajar el *device* mediante la sintaxis *<<< tamaño grid, tamaño bloque >>>* donde el primer parámetro indica el número de bloques en una malla y el segundo el número de hilos por bloque. Además de especificar el número de bloques y de hilos, también se pueden definir las dimensiones de los mismos mediante variables dimensionales (*dim3*), que contiene tres componentes (*x*, *y*, *z*), como se muestra en el código de ejemplo de la figura 2.18, donde se invoca un *kernel* definiéndose una malla

bidimensional de 2x2 y bloques tridimensionales de 4x2x2, lo que hace un total de 4 bloques de 16 hilos cada uno.

```
dim3 dimGrid(2, 2, 1);

dim3 dimBlock(4, 2, 2);

kernel <<< dimGrid, dimBlock >>> (d_A, d_B, d_C);
```

Figura 2.18. Ejemplo de lanzamiento de un kernel en CUDA.

Cada hilo que se ejecuta en un *kernel* posee un identificador que es accesible dentro del *kernel* por medio de una variable *build-in*, llamada *threadIdx*. Este identificador puede ser unidimensional, bidimensional o tridimensional dependiendo de la dimensión del bloque.

Los hilos en una malla se organizan, como se muestra en la figura 2.19. En donde, en un primer nivel, los bloques están organizados en una malla de una, dos o tres dimensiones, y en un segundo nivel, los hilos están organizados en bloques también de hasta tres dimensiones. Todos los bloques en una malla alojan el mismo número de hilos. Cada bloque está identificado a través de una variable *blockIdx* y la dimensión de cada bloque es accesible mediante la variable *blockDim*.

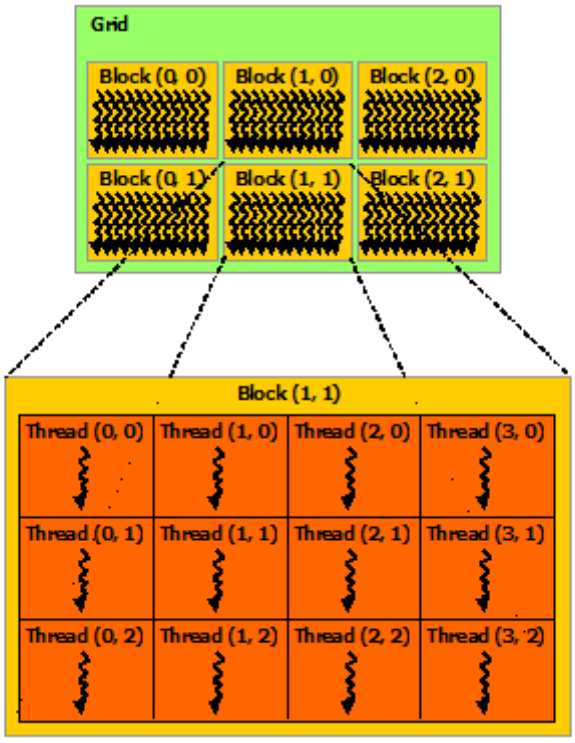

Figura 2.19. Estructura de una malla de bloques de hilos [32].

Existe un número máximo de hilos por bloque restringido por los recursos de memoria disponible en cada núcleo del procesador. En las GPUs actuales este límite es de 1024 hilos por bloque. Sin embargo, un *kernel* puede ejecutar numerosos bloques de hilos de la misma dimensión y tamaño, multiplicándose el número de hilos que pueden ser lanzados.

Los hilos pertenecientes a un mismo bloque son capaces de cooperar y compartir datos a través de la memoria compartida (*shared memory*) y pueden sincronizarse para coordinar los accesos a memoria mediante la función de barrera de sincronización *__syncthreads()*. Cuando se realiza una llamada a esta función, los hilos de un mismo bloque esperan a que todos alcancen ese punto.

2.4.2. Modelo de memoria

En CUDA, tanto la memoria de la CPU como la memoria de la GPU son independientes entre sí, pudiéndose transferir memoria de uno a otro.

Uno de los primeros pasos en la ejecución de un programa en CUDA es la asignación de memoria dinámica en la GPU y el traspaso de los datos necesarios desde el *host* al *device*, para posteriormente poder ejecutar un *kernel* en la GPU. Al finalizar la ejecución en el *device*, se transfieren los resultados obtenidos al *host* y se debe liberar la memoria asignada en la GPU. Para ello, se disponen de una serie de funciones (APIs) que nos permiten realizar todas estas operaciones, como se verá más adelante en el apartado 2.4.3.

2.4.2.1 Tipos de memoria CUDA

Un dispositivo CUDA posee diferentes tipos de memoria a las que tiene acceso la aplicación. Cada tipo de memoria tiene distintas propiedades como la latencia, el alcance y la duración, que hay que tener en cuenta en el diseño del algoritmo para aumentar la eficiencia en la ejecución.

Los distintos tipos de memoria son: registros, memoria local, memoria compartida memoria global y memoria constante, como se puede observar en la figura 2.20.

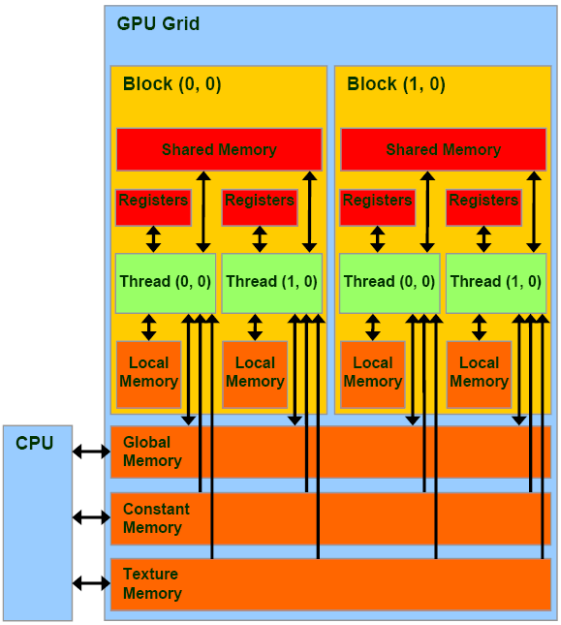

Figura 2.20. Modelo de memoria CUDA [27].

A continuación se explican las características principales de cada uno de los tipos de memoria utilizados en la programación en CUDA [27]:

Registros

Las variables escalares declaradas en una función *kernel* son almacenadas por defecto en los registros. Los arrays de tipo constante también son almacenados como registros.

Los registros son asignados a los hilos de forma individual, de manera que, cada hilo dentro de un bloque tendrá una versión propia de las variables almacenadas como registros y sólo podrá acceder a sus propios registros. La existencia de los registros viene determinada por la existencia del hilo, por lo que, cuando un hilo finaliza su ejecución, nunca se podrá acceder al registro de nuevo. El acceso a los registros es muy rápido, pero el número de registros disponibles por cada bloque está limitado.

Las variables almacenadas como registros son tanto de lectura como de escritura sin necesidad de ser sincronizadas.

Memoria local

Aquellas variables dentro de un *kernel* que no pueden ser almacenadas como registros lo harán como memoria local. A diferencia de los registros, la memoria local presenta una latencia mayor.

La memoria global, como ocurre con los registros, es privada de cada hilo, por lo que ningún otro hilo puede acceder a las variables almacenadas en memoria local. Su vida se corresponde con la del hilo.

Las variables en memoria local son tanto de lectura como de escritura sin necesidad de sincronización.

Memoria compartida

Las variables declaradas dentro del *kernel* con el atributo *__shared__* son almacenadas en memoria compartida. Todos los hilos dentro de un mismo bloque pueden acceder a datos almacenados en la memoria compartida. La vida de la memoria compartida corresponde con la vida del bloque, de manera que, cuando un bloque ha finalizado su ejecución, ya no se podrá acceder a la memoria compartida.

El acceso a la memoria compartida es muy rápido (unas 100 veces más rápido que la memoria global) aunque la cantidad de memoria compartida es limitada. La memoria compartida se usa para sustituir parte de la memoria global que se usa con frecuencia para aumentar la velocidad de ejecución de las aplicaciones.

Las variables almacenadas en memoria compartida son tanto de lectura como de escritura. Los hilos de un bloque deben ser sincronizados si se requiere acceder a datos de la memoria compartida que será leída o escrita por otros hilos dentro del mismo bloque. La sincronización de los hilos de un bloque se realiza mediante la función *__syncthreads()*.

Memoria global

Las variables precedidas por la palabra *__device__* en la declaración son almacenadas en la memoria global de la GPU. El acceso a este tipo de memoria es lento. Sin embargo, la capacidad de la memoria global es mucho mayor que la de la memoria compartida.

Al contrario de lo que ocurre con los anteriores tipos de memoria mencionados anteriormente, la memoria global puede ser leída y escrita tanto por el *host* a través de la función *cudaMemcpy* como por el *device*. La memoria global se declara desde el *host* con la función *cudaMalloc* y se libera también en el *host* usando *cudaFree*.

El contenido de la memoria global persiste durante toda la aplicación y todos los hilos de todos los *kernels* pueden acceder a ella. Sin embargo, se debe tener en cuenta que no es posible sincronizar los hilos de diferentes bloques. Por lo que, la única

manera de asegurar que la memoria global está sincronizada es esperando a que el *kernel* finalice su ejecución y lanzar otro *kernel* de nuevo.

Memoria constante

Las variables almacenadas en memoria constante van precedidas de la palabra *__constant__*. Se declaran fuera del espacio de las funciones. Al igual que en la memoria global, la memoria constante es accesible desde todos los hilos de todos los *kernel* y persiste durante toda la aplicación. Sin embargo, presenta una capacidad mucho menor que la memoria global.

El acceso a este tipo de memoria es más rápida que a la memoria global ya que la memoria constante es alojada en la caché de la memoria global. No puede ser escrita desde el *device*, en cambio, el *host* sí puede escribir y leer usando las funciones *cudaMemcpyToSymbol* y *cudaMemcpyFromSymbol* respectivamente.

En la tabla 2 se muestra un resumen de las características de cada una de los distintos tipos de memoria.

Tipo de memoria	Declaración	Ámbito	Duración	Latencia
Registros	Variables automáticas escalares	Hilo	Kernel	Baja
Local	Variables automáticas arrays	Hilo	Kernel	Alta
Compartida	**__shared__** int Var	Bloque	Kernel	Baja
Global	**__device__** int Var	Malla	Aplicación	Alta
Constante	**__constant__** int Var	Malla	Aplicación	Baja

Tabla 2. Características de los tipos de memoria en CUDA [28].

2.4.2.2 La memoria como limitador de la eficiencia

Para conseguir una mayor eficiencia en la ejecución de los algoritmos se debe evitar, en la medida de lo posible, los accesos a memoria global. En su lugar se pueden usar registros, memoria compartida o constante, que son de acceso mucho más rápido. Sin embargo, la cantidad de memoria disponible en el dispositivo limita el número de hilos y de bloques que pueden ser alojados en los multiprocesadores [28].

Cuanta más memoria compartida por bloque se requiere, menor número de bloques podrán ser utilizados en el multiprocesador. Lo mismo ocurre con la cantidad de registros por bloque, limitando el número de hilos que pueden ser utilizados en el multiprocesador.

2.4.3. CUDA Runtime API

La herramienta de CUDA (*NVIDIA CUDA Toolkit*) proporciona una serie de funciones que permiten el desarrollo de las aplicaciones CUDA. Estas funciones, entre otros muchos usos, permiten el control del dispositivo, de los hilos y de la memoria[33]. Entre las más habituales se encuentran:

Función	Descripción
cudaMalloc(void ** devPtr, size_t size)	Asignación de memoria global en el dispositivo.
cudaFree(void * devPtr)	Liberación de memoria del dispositivo.
cudaMemcpy(void * hostPtr, void * devPtr, size_t size, cudaMemcpyHostToDevice)	Copia de memoria desde el host a memoria global en el device.
cudaMemcpy(void * devPtr, void * hostPtr, size_t size, cudaMemcpyDeviceToHost)	Copia de memoria global desde el device a host.
cudaMemset(void * devPtr, int value, size_t count)	Asignación del valor constante *value* a la memoria.
cudaThreadSynchronize(void)	Espera hasta que el dispositivo haya completado todas las tareas programadas.

2.4.4. Optimización en CUDA

La optimización del código es esencial para mejorar el rendimiento de una aplicación y es, probablemente, una de las tareas más costosas en la aceleración de algoritmos en GPU.

Para que una aplicación pueda ser acelerada mediante GPU debe cumplir los siguientes requisitos [29]:

- El algoritmo debe ser masivamente paralelo, pudiéndose dividir en tareas independientes.

- El tiempo de ejecución del algoritmo sin acelerar debe ser considerablemente mayor que los tiempos de transferencia de memoria entre la CPU y la GPU.

- El tamaño del *kernel* debe ser limitado. Se debe poder alojar un número de hilos elevado sin llegar a superar la capacidad de memoria disponible en el *device*.

Conseguir un código optimizado no resulta sencillo, y puede llevar gran parte del trabajo en la aceleración de algoritmos. Algunas de las estrategias recomendadas se presentan a continuación [29]:

1. En las aplicaciones CUDA, la transferencia de memora entre el *host* y el *device* son unas de las instrucciones que requieren mayor tiempo computacional, por lo que se deben minimizar dichas transferencias. A veces es conveniente ejecutar varios *kernels* en la GPU, aunque presenten un menor rendimiento que en la CPU, con la finalidad de evitar las transferencias de memoria. También, es aconsejable copiar memoria entre CPU y GPU de una gran cantidad de datos en lugar de hacer pequeñas transferencias múltiples.

2. Debido a la elevada latencia que presenta la memoria global en comparación con la memoria compartida, se debe utilizar ésta en su lugar con el objetivo minimizar el acceso a la memoria global. Sin embargo, para beneficiarnos de la baja latencia de la memoria compartida, el algoritmo debe requerir un acceso frecuente a memoria global.

3. En la mayoría de los algoritmos existen partes que deben ejecutarse secuencialmente, pero que admiten algoritmos de reducción que permiten la ejecución paralela de algunas de las instrucciones. Algunos de los usos de estos algoritmos de reducción son, por ejemplo, calcular la suma o hallar el máximo o mínimo de los elementos de un vector. Los dos tipos de reducciones más habituales son las reducciones binarias y las reducciones atómicas [30]:

    - Las **reducciones binarias** consisten en combinar una secuencia de N elementos de un vector en un valor único, aplicando algún operador. Calculando varias operaciones en paralelo, se obtiene el resultado en $\log_2(N)$ pasos. Un ejemplo de reducción binaria para la operación suma se muestra en la figura 2.21.

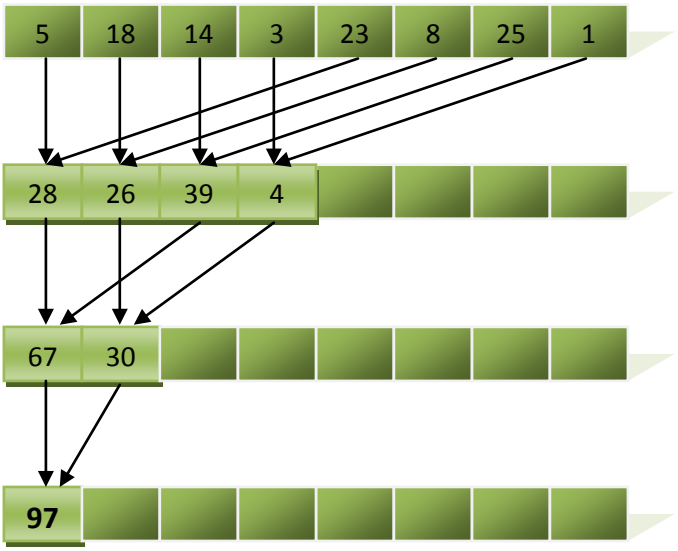

Figura 2.21. Esquema de algoritmo de reducción para la suma de dos vectores.

- Las **reducciones atómicas** se realizan mediante el uso de funciones atómicas que permiten coordinar operaciones sobre memoria global y memoria compartida. Las funciones atómicas garantizan que se opere sin que haya interferencia entre hilos que acceden a una misma variable. Por ejemplo, la función atómica que se muestra a continuación permite calcular de forma paralela la suma de todos los elementos de un vector *v* y guardar el resultado en la variable *sum*.

```
atomicAdd(&sum,v[i]);
```

4. Evitar la divergencia en el flujo de instrucciones dentro de un mismo *warp*.

5. El número de hilos por bloque debe ser múltiplo de 32 para que el sistema ofrezca una mayor eficiencia.

6. Utilizar la biblioteca rápida de matemáticas (*-use_fast_math*) siempre que la rapidez compense la precisión.

7. Utilizar, cuando sea posible, operaciones a nivel de bit en lugar de operaciones aritméticas. Por ejemplo, utilizar el operador de rotación de bits >> si se quiere dividir entre un número múltiplo de 2 o el operador & para hallar el resto de una división [31].

$$\frac{i}{n} \rightarrow i \gg \log_2(n)$$

$$i\%n \rightarrow i \,\&\, (n-1)$$

2.5. Algoritmo de matching

Como ya se ha mencionado anteriormente, el algoritmo de *matching*, en el reconocimiento biométrico del iris, permite la comparación de patrones de iris para decidir si los patrones comparados pertenecen al mismo ojo. En el presente trabajo se ha utilizado el método de la distancia de *Hamming*, que indica la diferencia entre dos códigos binarios.

Para hallar la distancia de *Hamming*, *HD*, se calcula el número de bits que difieren de un código binario a otro empleando el operador lógico *XOR*, cuyo valor es 1 cuando dos bits son diferentes. La distancia de *Hamming* se define como la suma de bits que difieren sobre el número total de bits, como se muestra en la siguiente expresión [34]:

$$HD = \frac{1}{N}\sum_{i=0}^{N} T1_i (XOR) T2_i$$

Donde N es el número total de bits en el patrón de iris, y $T1$ y $T2$ son las plantillas (obtenidas en la etapa de codificación del iris) de los patrones que se quieren comparar.

Cuanta mayor similitud exista entre dos patrones, la distancia de *Hamming* tendrá un valor más próximo a 0. Idealmente, la comparación de patrones pertenecientes al mismo iris daría como resultado 0, ya que ambos tendrían el mismo patrón y no habría diferencia entre ellos. Sin embargo, en la práctica esto no ocurre debido a las imperfecciones producidas en los pasos anteriores de captura, segmentación y normalización del iris. Estas imperfecciones pueden deberse a reflejos o a ruidos en la imagen del iris no detectados. Si los patrones son distintos, el valor de la distancia de *Hamming* esperado es 0.5, ya que, estadísticamente, la independencia implica que ambos patrones de bits sea totalmente aleatoria, y por tanto, la probabilidad de que cada uno de los bits coincida con el mismo bit de otro patrón distinto es del 50%.

De la etapa de codificación, además de obtenerse los códigos del iris, también se obtienen las máscaras de ruido que han de tenerse en cuenta a la hora de calcular la distancia de *Hamming*. Aquellos bits de las máscaras de ruido cuyo valor es 0, se considerarán bits significativos, que se utilizarán en el cálculo de la distancia de

Hamming. La nueva expresión incluyendo las máscaras de ruido queda de la siguiente manera[34]:

$$HD = \frac{\sum_{i=0}^{N}[T1_i(XOR)T2_i(AND)NOT(M1_i)(AND)NOT(M2_i)]}{N - \sum_{j=0}^{N}[M1_j(OR)M2_j]}$$

Donde $M1$ y $M2$ son las máscaras de ruido asociadas a las plantillas $T1$ y $T2$ respectivamente.

En el algoritmo de *matching* también ha de solucionarse el problema de la rotación del ojo, corrigiendo las desalineaciones en la normalización del iris producidas por las diferencias de rotaciones entre las imágenes capturadas. El método propuesto por Daugman consiste en rotar los bits de una de las plantillas en dirección horizontal, hacia la izquierda y hacia la derecha, calculando el valor de la distancia de *Hamming* para cada rotación. El número de rotaciones de bits corresponde a una rotación real del iris de $2°$. El número de rotaciones requeridas para corregir las desalineaciones en la normalización estará determinada por la mayor diferencia entre ángulos entre dos imágenes de iris.

A continuación se muestra un ejemplo simplificado del algoritmo de *matching* para dos códigos binarios de 10 bits, en donde se realiza una rotación de 2 bits hacia la izquierda y hacia la derecha:

T1 =	0	1	0	1	1	1	0	0	1	1
T2 =	1	1	0	1	0	1	1	1	0	0
T1(XOR)T2 =	1	0	0	0	1	0	1	1	1	1

$HD = 0.6$

Desplazando *T1* dos bits hacia la izquierda:

T1 =	0	1	1	1	0	0	1	1	0	1
T2 =	1	1	0	1	0	1	1	1	0	0
T1(XOR)T2 =	1	0	1	0	0	1	0	0	0	1

$HD = 0.4$

Desplazando *T1* dos bits hacia la derecha:

T1 =	1	1	0	1	0	1	1	1	0	0
T2 =	1	1	0	1	0	1	1	1	0	0
T1(XOR)T2 =	0	0	0	0	0	0	0	0	0	0

$\boldsymbol{HD = 0.0}$

En el primer caso, sin aplicar la rotación de bits, la distancia de *Hamming* indica que ambos códigos binarios difieren en un 60%, distancia demasiado grande como para considerar que ambos códigos pertenecen al mismo iris. En el segundo caso, se han desplazado dos bits hacia la izquierda, obteniendo una distancia de *Hamming* menor a la anterior, es decir, la similitud entre ambos códigos es mayor. Por último, aplicando una rotación de 2 bits hacia la derecha, se obtiene la menor de las distancia de *Hamming* calculadas, ésta será la usada para decidir si ambos códigos pertenecen al mismo iris, que en este caso se trata del mismo código binario.

3. PARALELIZACIÓN DEL ALGORITMO

En la implementación del algoritmo de *matching* paralelizado, se han realizado diferentes modelos o formas de paralelización hasta llegar al algoritmo definitivo con mayor aceleración conseguido.

El tipo de paralelismo principal en el algoritmo es el paralelismo a nivel de datos, ya que se deben realizar las mismas operaciones sobre un conjunto de datos.

Como se ha visto anteriormente, el algoritmo de *matching* compara varias veces, utilizando el método de la distancia de *Hamming*, dos códigos binarios haciendo una rotación de bits para corregir las desalineaciones producidas por la rotación del ojo, cogiendo la menor de entre todas las distancias de *Hamming* calculadas.

3.1. Cálculo de la distancia de Hamming

La diferencia entre dos patrones de iris se calcula empleando la distancia de *Hamming* siguiendo la ecuación 3.1:

$$HD = \frac{\sum_{i=0}^{N}[T1_i(XOR)T2_i(AND)NOT(M1_i)(AND)NOT(M2_i)]}{N - \sum_{j=0}^{N}[M1_j(OR)M2_j]} \quad \text{(EC 3.1)}$$

Donde N es el número total de bits en el patrón de iris, $T1$ y $T2$ son las plantillas de los patrones que se quieren comparar y $M1$ y $M2$ son las máscaras de ruido asociadas a cada una de las plantillas.

Haciendo la equivalencia $NOT(M1_i)(AND)NOT(M2_i) = NOT(M1_i(OR)M2_i)$, y sustituyendo $\tilde{T} = T1(XOR)T2$ y $\tilde{M} = M1(OR)M2$, la expresión de la distancia de *Hamming* queda como sigue:

$$HD = \frac{\sum_{i=0}^{N}[\tilde{T}_i(AND)NOT(\tilde{M}_j)]}{N - \sum_{j=0}^{N}[\tilde{M}_j]} \quad \text{(EC 3.2)}$$

$$HD = \frac{\sum_{j=0}^{N} A}{N - \sum_{j=0}^{N} B} = \frac{N\acute{u}mero\ de\ bits\ distintos}{N\acute{u}mero\ total\ de\ bits\ significativos} \quad \text{(EC 3.3)}$$

En la aplicación implementada, se parte de los datos adquiridos en *Matlab* tras el procesado de las imágenes de ojos hasta la etapa de codificación, donde se obtiene el *iriscode* una vez extraídas las características de la textura del iris. Por cada una de las imágenes se obtienen las plantillas y las máscaras de ruido asociadas a cada ojo. Estos datos son representados mediante matrices binarias de 480x20, sin embargo, como se verá más adelante en el apartado de implementación y desarrollo, se han agrupado los 20 elementos de cada una de las columnas de las matrices binarias en números enteros de 32 bits con el fin de ahorrar memoria RAM y optimizar la aplicación. De manera que, los datos procesados en el algoritmo de *matching* son representados mediante vectores de 480 números enteros de ancho de palabra 32 bits, de los cuales, los 12 bits más significativos son de valor cero, por lo que solo interesan los 20 bits menos significativos. Cada uno de los números contenidos en el vector serán procesados paralelamente, en la medida de lo posible. De este modo, las operaciones XOR, OR y AND se pueden aplicar a los 480 elementos de forma simultánea.

Una vez realizadas estas operaciones lógicas, se deben obtener el número de bits distintos entre las plantillas y el número total de bits significativos, es decir, aquellos bits cuyo valor es 0 en ambas máscaras de ruido. Para ello, se deben contar los bits de valor 1 presentes en los vectores resultantes denotados como A y B en la ecuación 3.3. Esto se ha dividido en dos etapas. En primer lugar, han de obtenerse el número bits de valor 1 de cada uno de los 480 elementos del vector y posteriormente, sumarlos todos.

1. Para la primera de las etapas se han implementado dos alternativas, en ambas se realizarán las operaciones sobre todos los elementos de los vectores en paralelo y se almacenarán para sumarlos en la siguiente etapa. A continuación se muestran las dos alternativas para la primera etapa:

   a) Al tratarse de números enteros, si se quiere hallar el número de bits de valor 1 para cada uno de los elementos del vector, se debe descomponer cada número decimal en un número binario, esto es, dividir el número decimal entre dos repetidas veces y sumar los restos de las divisiones.

      Como los números de ambos vectores (A y B) son de 20 bits, como máximo serán necesarias 20 divisiones para cada uno de los elementos. Todas las operaciones se realizarán para todos los elementos en paralelo y se almacenarán para sumarlos posteriormente.

      A pesar de que se actúe de forma paralela sobre todos los elementos del vector, el inconveniente de este método es que es, en parte, secuencial, ya que son necesarias varias iteraciones hasta descomponer los 20 bits del número decimal.

En el siguiente diagrama de flujo, se representa el algoritmo utilizado, donde nA y nB son vectores que contendrán la suma de bits de valor 1 de cada uno de los elementos de los vectores A y B. Todos los elementos del vector indicados con el índice " i " en el diagrama de flujo son procesados paralelamente.

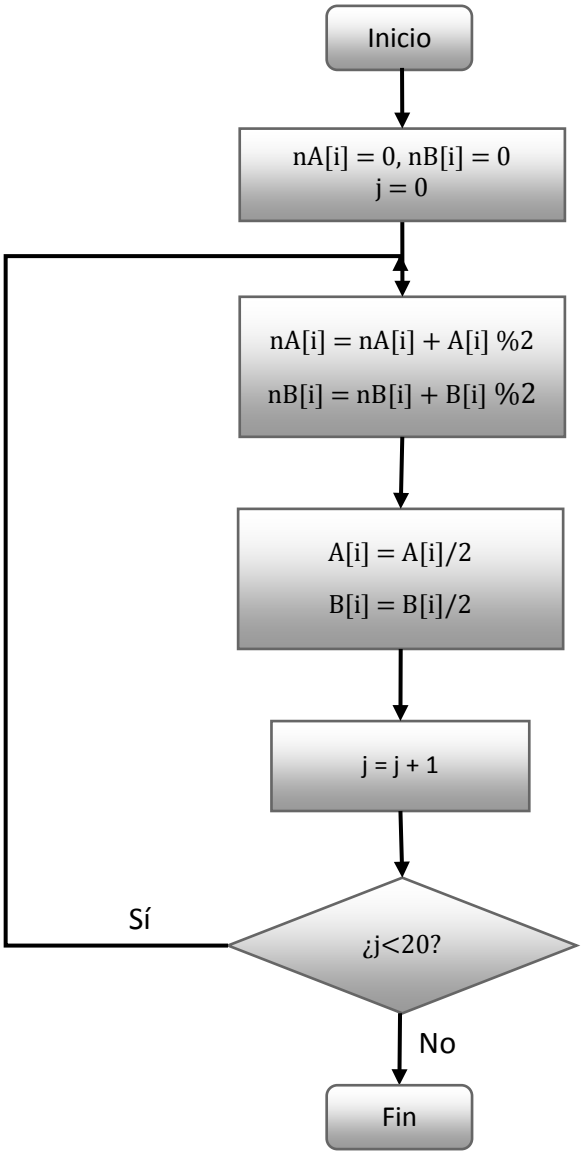

Figura 3.1. Diagrama de flujo de la suma de bits de valor 1 contenidos en los vectores A y B de números decimales.

b) Otra alternativa para calcular el número de bits de valor 1 es operar a nivel de bit sobre todo el conjunto de datos de forma simultánea. Este método, mostrado en la figura 3.2, requiere menos instrucciones que el anterior ya que no es necesario realizarse en varias iteraciones.

```
nA[i] = A[i] - ( ( A[i] >> 1 ) & 0x55555555 );
nA[i] = ( nA[i] & 0x33333333 ) + ( ( nA[i] >> 2 ) & 0x33333333);
nA[i] = ( ( nA[i] + ( nA[i] >> 4 ) & 0x0F0F0F0F ) * 0x01010101) >> 24;
```

Figura 3.2. Algoritmo alternativo para calcular el número de bits de valor 1 contenidos en un vector de números enteros.

Como se puede observar en el algoritmo de la figura anterior, se usan una serie de números expresados en base hexadecimal que representan los siguientes números en binario:

```
0x55555555 = 01010101010101010101010101010101
0x33333333 = 00110011001100110011001100110011
0x0F0F0F0F = 00001111000011110000111100001111
```

Figura 3.3. Representación binaria de los números utilizados.

Cada uno de estos números, expresados en forma binaria, está formado por 16 unos y 16 ceros alternados de forma distinta como se muestra en la figura 3.3.

Para una mejor comprensión del algoritmo de la figura 3.2, éste se puede expresar en cuatro pasos de la siguiente forma:

```
1. nA[i] = (A[i]&0x55555555) + (( A[i] >> 1 ) & 0x55555555);
2. nA[i] = (nA[i] & 0x33333333) + ((nA[i] >> 2) & 0x33333333);
3. nA[i] = (nA[i] & 0x0F0F0F0F) + ((nA[i] >> 4) & 0x0F0F0F0F)
4. nA[i] = (nA[i] * 0x01010101) >> 24;
```

Figura 3.4. Algoritmo para calcular el número de bits de valor 1 contenidos en un vector de números enteros expresado en mayor número de pasos.

En el primer paso se divide el número entero de 32 bits en 16 grupos de dos bits cada uno, y se cuenta el número de bits de valor 1 en cada uno de los grupos de pares de bits. Un conjunto de dos bits como máximo puede tener sus dos bits con valor 1, por lo que la cuenta se puede codificar en un número de dos bits. La primera instrucción es la suma de dos partes, en el primer sumando (A&0x55555555) se conservan los bits de valor uno en posición impar y en el segundo ((A>>1) & 0x55555555) se conservan los unos en posición par y colocan los unos pares en posiciones impares.

De forma análoga, después de los dos siguientes pasos se obtiene un número de 32 bits que puede agruparse en cuatro números de 8 bits cuyo valor es la cuenta de bits de valor 1 en el conjunto de 8 bits.

En el cuarto paso se multiplica el número obtenido por 0x01010101, que dará como resultado otro número cuyos 8 bits más significativos contienen la cuenta de bits de valor 1 que se desea obtener.

Por último, realizando una pequeñas optimizaciones el algoritmo se puede expresar como se muestra en la figura 3.2.

2. Una vez completada esta primera etapa, se han de sumar todos los elementos del vector obtenido en el primer paso, con el fin de hallar el total de números de valor 1 del vector completo. Para ello se han implementado otras dos alternativas:

a) Uno de los métodos implementados consiste en la reducción binaria. Esta técnica permite reducir el número de iteraciones necesarias para sumar N elementos de un vector. En la primera iteración se divide el vector en dos mitades y se aplica el operador suma simultáneamente sobre pares disjuntos del vector quedando $\mathrm{N}/2$ elementos sobre los que operar en la siguiente iteración. De manera que, el número de iteraciones necesarias para un vector de tamaño N será $\log_2(\mathrm{N})$. Cada iteración depende del resultado del anterior, por lo que es necesario realizar una sincronización entre iteraciones para llevar un orden entre los accesos a memoria. Véase la figura 2.21 del apartado 2.4.4 en la que se muestra un ejemplo del algoritmo de reducción.

b) Como alternativa al método anterior, se puede hacer uso de la función atómica *atomicAdd* que proporciona la API de CUDA. Ésta permite realizar la suma de todos los elementos del vector en paralelo de forma sincronizada.

En busca de un algoritmo optimizado, de entre estas dos alternativas para cada una de las etapas mencionadas para hallar el número total de bits con valor 1 contenidos en un vector de números enteros, se ha observado que las que presentan un mayor rendimiento en la aplicación son la opción b) en el caso de la primera etapa, y la opción a) para la segunda. En el presente trabajo únicamente se dan resultados de estas dos opciones mencionadas con los que se obtienen los mejores rendimientos.

Como era de esperar, para la primera etapa la opción b) es la más optima debido a que no es necesario realizar sucesivas iteraciones de manera secuencial como sucede en la opción a), en donde se requieren hasta 20 iteraciones. Además, las operaciones realizadas son en su mayoría operaciones a nivel de bits que presentan un mayor rendimiento respecto a las operaciones aritméticas.

Para la segunda etapa, la reducción binaria resulta más óptima que las operaciones atómicas en el algoritmo implementado. Los algoritmos de reducción presentan un mayor rendimiento cuando son aplicados sobre un conjunto elevado de

datos, como es el caso de la presente aplicación, en la que se trabaja con vectores de 480 elementos. Mientras que las operaciones atómicas son más eficientes para un número reducido de datos.

3.2. Rotación de bits

La rotación de bits corrige las posibles diferencias entre ángulos de dos patrones de iris. Se debe realizar un desplazamiento horizontal de bits hacia la izquierda y hacia la derecha tanto en la plantilla como en la máscara de ruido asociada, ambos representados mediante vectores.

El desplazamiento horizontal de bits es equivalente a un desplazamiento de los elementos del vector. Cada desplazamiento se realiza en dos partes, en la primera se copian $(N - b)$ elementos simultáneamente, y en la segunda se copian los b elementos restantes, siendo N el número de elementos que contiene el vector y b el número de bits que se desplazan. En la figura 3.5 se muestra un ejemplo en el que se realiza un desplazamiento de 2 bits hacia la izquierda.

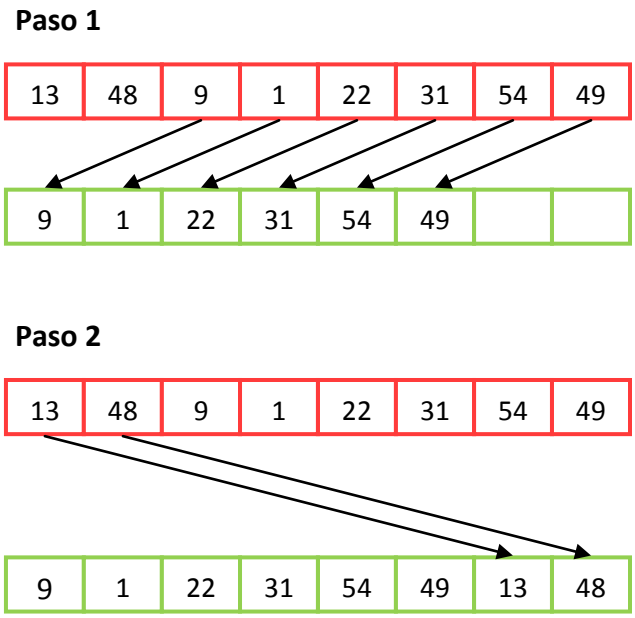

Figura 3.5. Esquema de permutación de los elementos de un vector. Representa a la rotación de 2 bits hacia la izquierda.

En el algoritmo implementado se realizan hasta 16 desplazamientos, de dos bits cada uno, 8 hacia la izquierda y 8 hacia la derecha. Esto implica que para una sola

comparación entre patrones se requiere el cálculo de la distancia de *Hamming* 17 veces, 16 para cada uno de los desplazamientos y 1 sin desplazamiento.

Además de la paralelización en el cálculo de la distancia de *Hamming* y en la rotación de bits, se puede ejecutar el algoritmo las 17 veces de forma simultánea y posteriormente escoger la menor de las distancias de *Hamming* obtenidas. Sin embargo, debido a que el objetivo es comparar un usuario frente a una base de datos para determinar su identidad en el que se requieren un elevado número de comparaciones, como por ejemplo para el control en un aeropuerto, como se verá más adelante, compensa realizar muchas comparaciones simultáneamente a cambio de calcular la distancia de *Hamming* varias veces de forma secuencial para una misma comparación.

4. IMPLEMENTACIÓN Y DESARROLLO

En este apartado se va a explicar el desarrollo del trabajo realizado para lograr el objetivo principal proyecto, que no es otro que la aceleración del algoritmo de *matching* mediante computación paralela sobre GPU.

El primer paso consiste en extraer los datos de entrada del algoritmo desde *Matlab* para posteriormente tratarlos con el objetivo de ahorrar memoria RAM de la aplicación, ya que, como se ha mencionado anteriormente, se trabaja con una elevada cantidad de datos. Seguidamente, se procesan los datos adquiridos mediante la implementación software del algoritmo de *matching* tanto de manera secuencial en lenguaje C, como paralela en CUDA, que se evaluarán y compararán sus rendimientos en apartado 5.

Por último, se mencionarán las herramientas software utilizadas para el desarrollo del trabajo.

4.1. Extracción y acondicionamiento de datos

Para la realización del proyecto se parte de una base de datos de imágenes de ojos (*CASIA-IrisV1*) y de una serie de ficheros en formato .m con el código fuente en *Matlab* de un sistema de reconocimiento biométrico basado en patrón de iris, desarrollado por Libor Masek de la Universidad de Australia Occidental (UWA) [35].

El código en *Matlab* consta de todas las etapas computacionales necesarias en un sistema de reconocimiento de iris partiendo de la captura de la imagen. En este proyecto se ha acelerado el algoritmo de *matching*, el cual corresponde con la última de las etapas. Por lo que se ha ejecutado el código *Matlab*, introduciendo las imágenes en formato .bmp, hasta la etapa de codificación, donde se obtiene el código del iris (*iriscode*) una vez extraídas las características de la textura del iris. Para cada una de las imágenes se extraen una plantilla y su máscara de ruido asociada de aquellas partes del iris afectadas por los párpados y las pestañas.

Tanto las plantillas como las máscaras de ruido, son un conjunto de datos binarios dispuestos en matrices de 20x480 que han sido extraídos desde *Matlab* mediante ficheros para implementar el algoritmo en lenguaje C.

El tamaño estándar de variables datos enteros en C es de 4 bytes, por lo que si se tiene en cuenta que por cada patrón de iris se requieren dos conjuntos de datos enteros de 20x480 y que se van a comparar un total de 756 iris, hacen un total de

14.515.200 datos enteros, que en memoria ocupa aproximadamente 55 MB. Para reducir el tamaño de los datos y por mayor simplicidad en la implementación del algoritmo, se ha decidido agrupar cada una de las columnas de la matriz de números binarios en conjuntos de 20 bits representados en números decimales, obteniéndose un vector de 480 elementos, de manera que el número de datos totales queda reducido a 725.760 datos enteros, ocupando un tamaño de 2,7 MB.

Además de la ventaja que esto supone en el ahorro de memoria, muy importante sobre todo en la implementación en CUDA, también mejora el rendimiento de la aplicación ya que se pueden procesar los datos con menor número de instrucciones en cualquier procesador de 32 bits o superior.

4.2. Implementación secuencial

Una vez extraídos los datos necesarios de cada uno de los patrones de iris y agrupados en conjuntos de 20 bits, se ha implementado el algoritmo de *matching* en código secuencial.

Para la implementación en serie, se ha decidido traducir el código *Matlab* a lenguaje C, con la diferencia principal de que en el algoritmo en *Matlab* los datos son tratados como conjuntos bidimensionales, mientras que en C, después de la agrupación de datos, se procesan conjuntos de datos unidimensionales.

Como ya se ha mencionado en el apartado anterior, el hecho de procesar los datos de forma agrupada, supone una ventaja tanto en el ahorro de memoria como en la eficiencia del algoritmo. Sin embargo, esto exige volver a descomponer los datos en bits para poder contar el número de bits con valor 1 en el cálculo de la distancia de *Hamming*.

Una vez implementado el algoritmo en lenguaje C, para asegurarse del correcto funcionamiento, se compararon los resultados obtenidos en C con los obtenidos en *Matlab*.

4.3. Implementación en CUDA

Como objeto de estudio del rendimiento de la aplicación acelerada, se han implementado distintos modelos o estrategias que varían en la forma de organización de hilos y *kernels* y en el número de iteraciones. El número de iteraciones consiste en el número de comparaciones entre patrones de iris realizadas, así este número difiere

dependiendo del modo de reconocimiento biométrico, ya sea verificación o identificación biométrica. En este apartado se explican las implementaciones realizadas del algoritmo de *matching* en CUDA.

Partiendo de la extracción de los datos mediante ficheros y el acondicionamiento de los mismos realizado en el *host*. El primer paso es la reserva de memoria global en el *device* y la transferencia de memoria entre el *host* y el *device* mediante funciones de la API de CUDA.

Para ahorrar el número de transferencias entre *host* y *device*, ya que se trata de un proceso lento, se almacenan todos los datos en dos vectores, uno que contiene todas las plantillas de iris de la base de datos y otro que contiene todas las máscaras de ruido asociadas, realizándose únicamente dos transferencias de memoria.

Modelo 1

En la comparación de dos patrones de iris se hallan 17 distancias de *Hamming* para cada una de las rotaciones de bits, de las que se debe seleccionar la menor de todas las distancias de *Hamming* calculadas.

En este modelo se lanza un primer *kernel*, donde se calculan todas las distancias de *Hamming* simultáneamente, y un segundo *kernel*, donde se comparan escogiendo la menor de las distancias de *Hamming* obtenidas.

El primero de los *kernels* presenta una malla unidimensional de 17 bloques y 512 hilos por bloque. En cada uno de los bloques se procesa el cálculo de la distancia de *Hamming* para una determinada rotación de bits. El número de hilos viene, en parte, determinado por el tamaño de los conjuntos de datos de cada patrón de iris, procesándose todos los elementos de cada vector de forma simultánea. Aunque estos conjuntos de datos son de 480 elementos, se lanzan 512 hilos por bloque debido a que el algoritmo de reducción binaria, utilizado en el cálculo de la distancia de *Hamming*, requiere que se realice sobre un conjunto de datos cuyo número de elementos sea potencia de 2. No obstante, también es posible aplicar la reducción binaria para un número de elementos arbitrario, sin embargo se observó que se consigue un menor rendimiento.

Como cada una de las distancias de *Hamming* se calculan de forma independiente en bloques distintos, ha de esperarse a que todos los hilos de cada bloque terminen su ejecución. Sin embargo, no es posible la sincronización entre bloques de un mismo *kernel*, por lo que se debe esperar a la finalización del primer *kernel* y lanzar un segundo *kernel*.

El segundo de los *kernels* consta de un único bloque de 32 hilos, en donde se comparan todas las distancias de *Hamming* utilizando un algoritmo de reducción binaria. Aunque el número de distancias de *Hamming* es menor de 32, por razones de eficiencia, es recomendable lanzar un número de hilos múltiplo de 32, ya que las instrucciones en la GPU se ejecutan en conjuntos de 32 hilos o *warps*.

Si el modo de reconocimiento es identificación biométrica, se deben realizar varias comparaciones entre todos los iris de la base de datos, en este caso, se repite el procedimiento lanzando varios *kernels* desde el *host* indicándose la posición de memoria en la que se encuentran cada uno de los patrones a comparar.

Modelo 2

En este modelo se obtienen cada una de las distancias de *Hamming* secuencialmente ejecutándose en el *device* un único *kernel* de un solo bloque de 512 hilos.

La ventaja de este modelo frente al anterior es que para un elevado número de comparaciones entre patrones de iris, se pueden lanzar tantos *kernels* independientes como comparaciones se requieran sin necesidad de esperar a que los *kernels* anteriores terminen su ejecución en la GPU.

Modelo 3

Este tercer modelo es una variante del modelo anterior diseñado para un elevado número de comparaciones entre distintos patrones de iris, en donde cada una de las distancias de *Hamming* independientes de cada comparación se calculan simultáneamente en bloques distintos de un mismo *kernel*. Por lo que el *host* lanza un *kernel* de un número de bloques igual al número de comparaciones y de 512 hilos por bloque. Sin embargo, si el número de bloques necesarios es muy elevado, se podría exceder la capacidad de memoria disponible en el *device*, por lo que se deben repartir las comparaciones en distintos *kernels*, esperando a que finalice la ejecución de un *kernel* en el *device* para lanzar otro nuevo.

4.4. Herramientas software

Para el desarrollo completo del Trabajo de Fin de Grado han sido necesarios varios productos software empleados para la implementación de las aplicaciones en serie y en paralelo.

Como ya se ha mencionado anteriormente, se ha hecho uso de *Matlab* para la extracción en ficheros de los datos de partida necesarios para la implementación del algoritmo.

Como entorno de programación se ha utilizado *Microsoft Visual Studio 2010* disponible para sistemas operativos Windows. Esta herramienta es muy empleada en la programación ya que lleva integrado un compilador y depurador (*debugger*), además soporta multitud de lenguajes de programación como *C/C++*, *Java*, *Python*, etc.

Para la implementación de aplicaciones en CUDA, NVIDIA proporciona todas las herramientas requeridas de forma gratuita en su página web. El software básico utilizado para Windows es *CUDA Toolkit 6.0* (actualmente existe una versión más reciente). Esta herramienta contiene un compilador para GPUs de NVIDIA, librerías, controladores de GPUs y herramientas para depurar y optimizar el rendimiento de las aplicaciones CUDA.

Para el caso de Windows, la herramienta de CUDA se configura automáticamente en una versión *Microsoft Visual Studio* compatible instalada en el sistema operativo, de manera que se puede desarrollar la aplicación en CUDA directamente desde *Visual Studio* utilizando el compilador de CUDA.

4.5. Características Hardware

Las características del hardware y de los procesadores utilizados influyen en el rendimiento de las aplicaciones que serán evaluadas en el siguiente aparatado. En la realización del trabajo se ha empleado un ordenador *HP 15 Notebook* con sistema operativo *Windows 8.1* (64 bits) y procesador *Intel Core i3-3110M* con velocidad de reloj de 2,4 GHz y 6 GB de memoria RAM. Como tarjeta gráfica, se ha empleado la tarjeta integrada en el ordenador que consiste en una *NVIDIA GeForce 820M* con frecuencia de reloj del núcleo de 719 MHz, que cuenta con 96 núcleos de CUDA y memoria total disponible de 3.018 MB.

Comparando las velocidades de la CPU y de la GPU, se puede observar que la frecuencia de reloj de la CPU es muy superior a la de la GPU, siendo más de tres veces más rápido.

5. EVALUACIÓN DEL RENDIMIENTO

Para la evaluación del rendimiento de la aplicación se han medido y comparado los tiempos de ejecución tanto en la implementación en serie como en la implementación paralela en CUDA.

Las mediciones de tiempo realizadas se han hecho para distinto número de repeticiones, en donde cada repetición significa una comparación entre dos patrones de iris de la base de datos. Así, simulando un sistema de reconocimiento biométrico, si el modo de reconocimiento consiste en la verificación de identidad, únicamente será necesaria una repetición, mientras que si se trata de identificación de identidad, se deberán comparar el patrón de iris del sujeto a identificar con el resto patrones de la base de datos. Debido a que la base de datos posee solamente 756 imágenes, para una evaluación del rendimiento más precisa, se ha elevado el número de repeticiones haciendo comparaciones entre todos los patrones con el resto de patrones disponibles en la base de datos.

5.1. Funciones de medición del tiempo

Para contabilizar el tiempo se ha hecho uso de las funciones *QueryPerformanceCounter()* y *QueryPerformanceFrecuency()* de la librería *"Windows.h"* en lenguaje C++. La primera función retorna el valor del contador en alta resolución (<1μs). Este valor de retorno debe convertirse a unidades temporales dividiendo por la frecuencia del contador obtenida de la segunda función. Estas funciones se han usado para obtener la duración de la implementación en serie y paralela.

CUDA, a su vez, también permite medir el tiempo de ejecución en la GPU, mediante el uso de funciones *CUDA Events* de la API. Estas funciones se han empleado para medir los tiempos de ejecución de los *kernels* así como de la reserva y transferencias de memoria de la GPU [33].

5.2. Resultados

La contabilización del tiempo se inicia una vez que se ha completado la lectura de todos los datos desde ficheros y se han agrupado en la CPU, y finaliza tras realizar todas las comparaciones entre patrones de iris oportunas.

En el caso de la implementación en CUDA, se han realizado distintas mediciones para evaluar el tiempo que transcurre en los procesos en los que interviene la GPU, como la asignación de memoria dinámica, las transferencias de memoria entre CPU y GPU y viceversa, y la ejecución de los *kernels* en cada una de las estrategias o modelos planteados en el apartado 4.3.

En primer lugar, se miden las duraciones de las aplicaciones tanto en serie como en paralelo para una sola comparación, en donde en la implementación en CUDA se ha medido únicamente el tiempo de cómputo paralelo, es decir, sin tener en cuenta la reserva y las copias de memoria de la tarjeta gráfica, obteniéndose los siguientes resultados:

	Serie	Modelo 1	Modelo 2	Modelo 3
Duración (ms)	0,42	0,25	0,32	0,32

Tabla 3. Duración del algoritmo de *matching* para una sola comparación en implementación secuencial y paralela para los tres modelos propuestos.

Nótese que para los modelos 2 y 3 de las implementaciones en CUDA se obtienen los mismos resultados, ya que para una sola comparación la disposición de hilos y bloques es idéntica en ambos casos.

Midiendo las duraciones de las transferencias de memoria entre *host* y *device*, y la reserva de memoria dinámica en el *device* para una sola repetición, se han obtenido los siguientes resultados:

Tareas	Tiempo (ms)
Reserva de memoria	0,25
Transferencia de memoria CPU-GPU	0,07
Transferencia de memoria GPU-CPU	0,04
Total	**0,36**

Tabla 4. Duraciones de las reserva de memoria y las transferencias de memoria de la targeja gráfica en CUDA para una sola comparación.

Como puede verse en la tabla 4, las transferencias de memoria entre CPU y GPU, y viceversa, junto con la reserva de memoria dinámica en la GPU, supone la mayor parte del tiempo total de la aplicación, superándose incluso las duraciones de los *kernels* mostradas en la tabla 3.

No obstante, en la práctica, si se mide el tiempo de la aplicación total empleando las funciones de la CPU, se observa que el valor de duración es mucho mayor que la

suma de los tiempos de las copias de memoria, reserva de memoria dinámica y la ejecución de los *kernels* en la GPU, siendo aproximadamente de unos 700 ms, tres órdenes más de magnitud de lo esperado. Esto se debe a que en la primera llamada a una función de la *runtime API* se inicializa el sistema de CUDA en el dispositivo, lo que lleva un cierto tiempo. Este tiempo es orientativo debido a que varía de unas ejecuciones a otras en varios milisegundos, estando alrededor de este valor.

Para el caso de un número de repeticiones más elevado, comparándose todos los patrones entre sí, en el que se deban reservar y transferir la memoria de todos los datos de las 756 imágenes pertenecientes a la base de datos, se han obtenido los siguientes resultados:

Tareas	Tiempo (ms)
Reserva de memoria	0,45
Transferencia de memoria CPU-GPU	1,72
Transferencia de memoria GPU-CPU	0,94
Total	**3,11**

Tabla 5. Duraciones de las reserva de memoria y las transferencias de memoria de la targeja gráfica en CUDA para varias comparaciones.

En este caso, tanto la reserva como las transferencias de memoria presentan una mayor duración, incrementándose en mayor medida las copias de memoria de la tarjeta gráfica.

Respecto a la duración del algoritmo completo, incluyéndose los tiempos de transferencia de reserva y transferencia de memoria, se han realizado varias medidas según el número de comparaciones y según los modelos de paralelización propuestos en el apartado 4.3, obteniéndose los resultados de la tabla 6 para las implementaciones en serie en CPU y en paralelo en GPU.

La siguiente tabla recoge los resultados obtenidos experimentalmente de las duraciones, expresadas en segundos, de las implementaciones tanto en serie como en paralelo. En la primera columna se indica el número de comparaciones de patrones de iris realizadas para cada una de las cuales se han medido los tiempos de ejecución. En la segunda columna se representan las medidas de tiempos obtenidas para la implementación secuencial en la CPU, y en las últimas tres columnas los tiempos de ejecución para los tres modelos de implementación paralela realizados en la GPU:

Comparaciones	CPU (s)	GPU-M1 (s)	GPU-M2 (s)	GPU-M3 (s)
100	0,041	0,713	0,711	0,716
250	0,103	0,758	0,773	0,726
500	0,206	0,828	0,842	0,772
750	0,308	0,883	0,928	0,808
1000	0,412	0,927	1,065	0,899
1509	0,641	2,223	1,172	0,941
2000	0,828	1,186	1,287	1,01
5000	2,058	1,834	2,089	1,446
10000	4,169	2,747	3,277	2,088
50000	20,694	11,062	12,748	6,813
100000	42,015	20,925	24,599	12,711
150000	62,557	30,024	36,444	18,607
200000	83,017	41,265	48,287	24,521
250000	103,934	51,367	60,133	30,422
285390	118,089	57,846	68,42	34,677

Tabla 6. Duraciones de las aplicaciones en serie y en paralelo para distintos números de comparaciones.

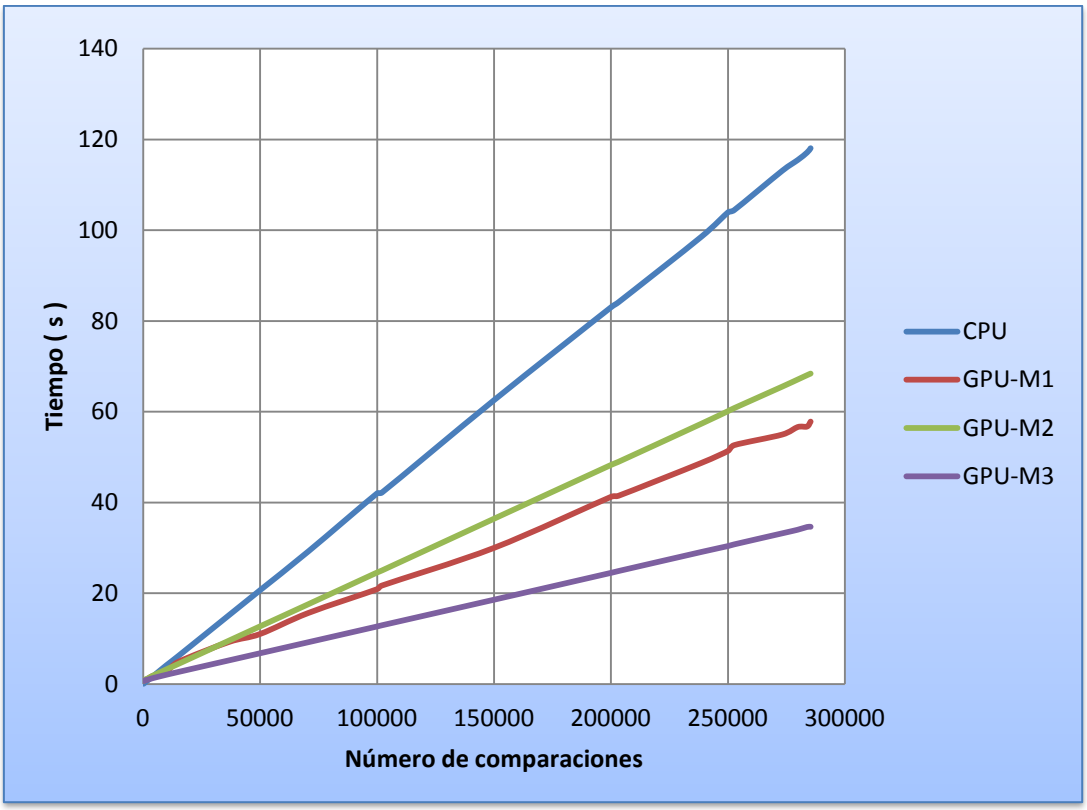

Figura 5.1. Representación gráfica de la duración de las implementaciones en serie y en paralelos en función del número de comparaciones realizadas.

En la figura 5.1 se han representado gráficamente los tiempos de ejecución frente al número de comparaciones de los resultados obtenidos (para una mayor precisión se han representado más puntos de los mostrados en la tabla 6). Se puede observar que para valores bajos de número de comparaciones existen intersecciones entres las curvas representadas. En la figura 5.2 se muestran únicamente los tiempos de ejecución de las aplicaciones para un número bajo de comparaciones con el objetivo de ver más claramente las intersecciones mencionadas:

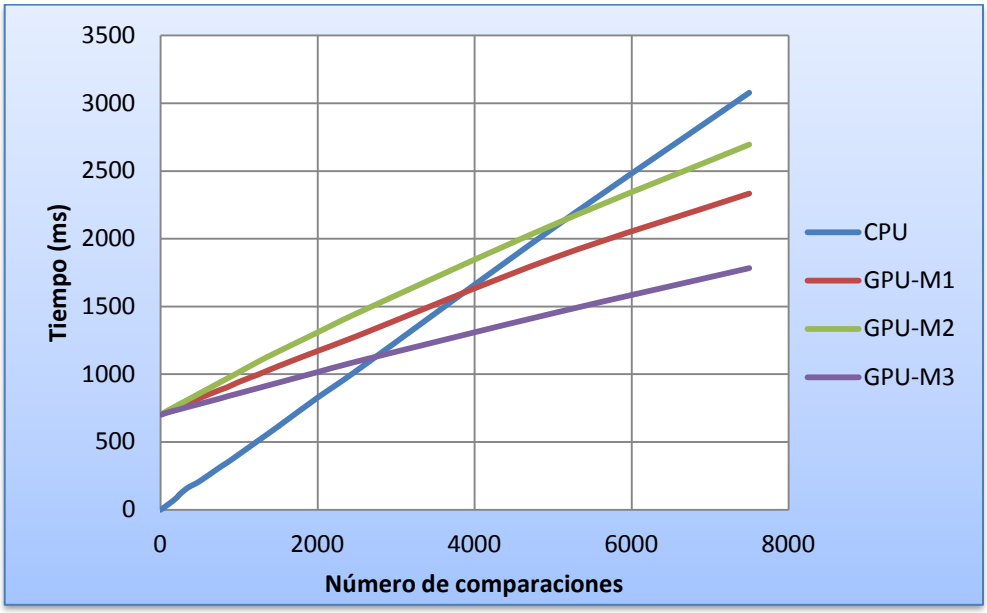

Figura 5.2. Intersecciones de las curvas de tiempos de ejecución.

Para la representación de la gráfica anterior se han medido las duraciones de las aplicaciones utilizando las funciones de CUDA mencionadas anteriormente en el apartado 5.1 que permiten medir el rendimiento. Se han empleado estas funciones con el objetivo de medir el tiempo de ejecución de la aplicación sin tener en cuenta el tiempo de inicialización del sistema CUDA, debido a que este tiempo varía notablemente de una ejecución a otra impidiendo una representación gráfica clara. No obstante, este tiempo sí se ha tenido en cuenta de manera teórica añadiendo 700 ms (duración aproximada de inicialización) al tiempo obtenido experimentalmente.

5.3. Discusión de resultados

Los resultados obtenidos en el apartado anterior indican una mejora considerable en la duración del algoritmo para cada uno de los modelos implementados en CUDA, cumpliéndose el objetivo principal del proyecto.

Analizando las duraciones obtenidas para el caso de una sola comparación representadas en la tabla 3, se ha conseguido acelerar el algoritmo reduciendo el tiempo de ejecución de la implementación en serie en un 40% en el primer modelo y en un 24% en el segundo y tercer modelo.

Sin embargo, si se tienen en cuenta los tiempos de copia y reserva de memoria en la tarjeta gráfica, el algoritmo implementado sobre la GPU ya no resulta rentable desde el punto de vista del rendimiento de la aplicación. Además, si también se tiene en cuenta el tiempo de inicialización del sistema CUDA la duración de la aplicación se incrementa drásticamente, por lo que para que una aplicación pueda ser acelerada en CUDA, debe presentar una duración mayor a la de los tiempos de reserva y transferencia de memoria y sobre todo, al tiempo de inicialización.

Como se puede observar en los resultados de las mediciones de tiempo obtenidos experimentalmente, representadas en la tabla 6, para un número bajo de comparaciones la implementación en serie en CPU es la mejor opción, mientras que para un número de comparaciones suficientes en el que la duración de la implementación en serie sobrepasa considerablemente los tiempos de inicialización, reserva y copias de memoria, la implementación en paralelo reduce notablemente la duración respecto a la secuencial.

Analizando la gráfica de la figura 5.2, se observa un comportamiento lineal en las curvas representadas. La curva correspondiente a la implementación en CPU (curva azul) presenta intersecciones con el resto de las curvas. Estas intersecciones indican el número mínimo de comparaciones necesarias para que sea rentable, en términos de rendimiento, la implementación de la aplicación en GPU.

Se ha realizado una regresión lineal para cada una de estas curvas con el objetivo de ajustar los datos a una recta y poder hallar analíticamente las intersecciones mencionadas. Las rectas de regresión obtenidas son las siguientes:

$$t_{CPU} = 0{,}4109 \cdot N + 5{,}002\ (ms)$$

$$t_{GPU-M1} = 0{,}2205 \cdot N + 716{,}52\ (ms)$$

$$t_{GPU-M2} = 0{,}2683 \cdot N + 734{,}24\ (ms)$$

$$t_{GPU-M3} = 0{,}1453 \cdot N + 711{,}31\ (ms)$$

Donde t_i son los tiempos de ejecución de las aplicaciones y N es el número de comparaciones.

	GPU-M1	GPU-M2	GPU-M3
Número de comparaciones	3737	5114	2660

Tabla 7. Número de comparaciones mínima para que sea rentable el uso de la GPU para cada uno de los modelos (corresponde con la intersección de las curvas de la figura 5.2).

De entre los tres modelos propuestos para la implementación en paralelo, el tercero es el que presenta un mayor rendimiento, llegándose a conseguir una aceleración **3,4** veces mayor, reduciendo el tiempo de la implementación en serie en un **70%** para una elevada cantidad de comparaciones.

La ventaja del modelo 3 frente a los otros dos se debe a que permite realizar múltiples comparaciones simultáneamente en bloques distintos reduciéndose el número de *kernels* lanzados respecto a los modelos 1 y 2. Aunque como se ha visto anteriormente, es necesario realizar una sincronización de bloques para no exceder la capacidad de memoria disponible en el *device*, el tiempo de lanzamiento de un *kernel* no es despreciable por lo que reducir el número de *kernels* supone una reducción de tiempo considerable.

6.CONCLUSIONES

Tras haber implementado el algoritmo de *matching* en CUDA y realizada la evaluación del rendimiento, visto en el apartado anterior, se ha obtenido una mejora notable del rendimiento frente a la implementación en serie sobre la CPU. Esta mejora del rendimiento se traduce en una reducción del tiempo de ejecución del algoritmo en hasta un 70%. Se ha cumplido el objetivo de reducir el tiempo de ejecución del algoritmo afirmándose así que con la aplicación de computación paralela se consiguen mejores rendimientos cuando se deben realizar un elevado número de cálculos sobre grandes cantidades de datos.

Como se ha visto en la implementación paralela, las formas de paralelizar un mismo algoritmo no son únicas, sino que existen multitud de caminos posibles que permiten llegar a los mismos resultados, y cada uno de ellos presenta un rendimiento distinto. En ocasiones, adaptar el algoritmo para la paralelización en CUDA implica realizar ciertos cambios sobre el algoritmo secuencial. Es tarea del programador buscar una de las posibilidades que ofrece CUDA en la programación y paralelización de los algoritmos de manera que se obtenga un mejor rendimiento.

La implementación de un algoritmo en CUDA no solo se ha basado en paralelizar el mismo sino que además ha sido necesario realizar una serie de optimizaciones sin las cuales el rendimiento de la aplicación era incluso muy inferior respecto a la implementación secuencial. En el desarrollo del presente trabajo se ha observado que las partes secuenciales penalizan notablemente el tiempo de ejecución del algoritmo paralelizado debido a que la velocidad de la GPU es mucho menor que la velocidad de la CPU.

Un aspecto a tener en cuenta en el rendimiento del algoritmo implementado en CUDA es el tiempo necesario para la reserva de memoria en la GPU y las transferencias de memoria entre la CPU y la GPU ya que son procesos costosos, pero imprescindibles, que se deben minimizar en la medida de lo posible. Se ha visto que para que un algoritmo pueda ser acelerado mediante su implementación en CUDA, éste debe presentar una duración en CPU superior a las duraciones de reserva y transferencias de memoria.

La aceleración conseguida de un algoritmo está estrechamente relacionada con las prestaciones de la tarjeta gráfica utilizada. Aplicaciones implementadas sobre distintas GPUs pueden presentar un diseño distinto por lo que los ejecutables no siempre son portables de una GPU a otra, sino que su implementación se debe realizar atendiendo a la arquitectura y a las prestaciones de la tarjeta empleada. Un factor muy importante en las prestaciones de la tarjeta gráfica para la aceleración de algoritmos

es la capacidad de memoria compartida, que limita la velocidad del algoritmo, junto con el número de hilos que se pueden alojar en un mismo bloque. Si bien las tarjetas gráficas de NVIDIA actuales pueden albergar hasta 1024 hilos por bloque, tarjetas más antiguas únicamente pueden albergar hasta 512 hilos por bloque.

7. PLANIFICACIÓN Y PRESUPUESTO

7.1. Planificación

Para la adecuada realización del Trabajo de Fin de Grado y el cumplimiento de los objetivos, se ha elaborado una planificación de las tareas de las que se compone el trabajo, estableciéndose un tiempo de duración estimado para cada una de ellas y dividiéndose en fechas aproximadas. Las planificación se ha dividido en 10 tareas:

- **Búsqueda de información (25 horas):** para la realización del trabajo es necesario una amplia documentación en programación en CUDA, reconocimiento biométrico, reconocimiento de iris y base de datos de imágenes de iris. Probablemente, la búsqueda de manuales de programación en CUDA es la más destacada en la elaboración del trabajo.

- **Aprendizaje (50 horas):** constituye una de las etapas más importantes del proyecto debido a la carencia de conocimientos previos. Ha sido necesario adquirir conocimientos en el campo de la biometría, prestando una mayor atención al reconocimiento de iris y al algoritmo de *matching* utilizado; conocimientos de programación en *Matlab* para la extracción de datos y conocimientos de programación en CUDA, en el que se incluye tanto el lenguaje de programación como el hardware de una GPU, que se debe tener presente en su programación.

- **Extracción y acondicionamiento de datos (20 horas):** esta tarea ha supuesto una parte importante en la elaboración proyecto, debido a que se ha requerido ejecutar en *Matlab* parte del algoritmo de reconocimiento de iris para la extracción de todos los datos, y crear una aplicación en C que lea los datos desde ficheros y los agrupe para mejorar la eficiencia del algoritmo a implementar.

- **Implementación en C (45 horas):** con todos los datos necesarios disponibles se implementó el algoritmo de *matching* en lenguaje C comprobándose su funcionamiento correcto.

- **Implementación en CUDA (50 horas):** una vez realizada la implementación en serie en lenguaje C, se puso en práctica los conocimientos adquiridos en CUDA para la paralelización del algoritmo proponiéndose varias alternativas.

- **Optimización (40 horas):** con el objetivo de mejorar el rendimiento de la aplicación en CUDA se ha hecho uso de algunas de las técnicas de optimización aprendidas. Este paso ha sido costoso y donde más se ha evolucionado ya que, en un primer momento, la aplicación en CUDA no ofrecía el rendimiento esperado, siendo incluso peor que en la implementación en serie.

- **Estudio del rendimiento (25 horas):** una vez implementada y optimizada la aplicación en CUDA, se ha realizado un estudio del rendimiento comparando la implementación en serie con las distintas implementaciones en paralelo realizadas, determinando la influencia del número de iteraciones en el rendimiento.

- **Desarrollo de la memoria (90 horas):** la redacción de la memoria ha sido la etapa que más tiempo ha requerido, en donde se exponen las bases teóricas para la comprensión del proyecto y los resultados obtenidos. Esta etapa se ha realizado paralelamente con otras tareas del proyecto.

- **Revisión (10 horas):** la revisión ha consistido en corregir errores en el texto y añadir algunas partes necesarias para obtener el documento completo que ha sido supervisado por el tutor.

El diagrama de *Gantt* de la figura 7.1 representa los meses en los que se han realizado cada una de las tareas del proyecto. Debido a la situación académica del autor, en los meses de periodo de exámenes, mayo y junio, se ha dedicado menor tiempo en el desarrollo del proyecto.

Figura 7.1. Diagrama de Gantt del Trabajo de Fin de Grado.

7.2. Presupuesto

Se ha desarrollado el presupuesto económico que supone la realización de este Trabajo de Fin de Grado. En él se incluyen los costos de personal, especificándose el número de horas empleadas, y de equipo técnico utilizado.

El coste de personal supone la mayor parte del presupuesto, en el que han intervenido el autor del trabajo, en calidad de Graduado en Ingeniería en Tecnologías Industriales, y el tutor, que ha dirigido y supervisado el proyecto completo, en calidad de Doctor en Ingeniería Industrial.

El resto del coste del presupuesto es debido a las herramientas software y hardware utilizadas en la elaboración del trabajo, donde se incluye las licencias de los softwares utilizados *Microsoft Visual Studio 2010* y *Matlab R2011a*, disponibles en la Universidad Carlos III de Madrid, y el ordenador con tarjeta gráfica NVIDIA empleado para la elaboración del trabajo y para la redacción de la memoria.

También se incluye la herramienta software *CUDA Toolkit 6.0* para el desarrollo de la aplicación en CUDA, que está disponible en Internet en la página oficial de NVIDA de forma gratuita.

Ítem	Descripción		Costes		
1	Personal	Doctor en Ingeniería	20 horas	40 €/h	800 €
2		Graduado en Ingeniería	355 horas	10 €/h	3.550 €
3	Herramientas software y hardware	Licencia Visual Studio 2010	550 €		
4		Licencia Matlab 2011	2.000 €		
5		Ordenador con tarjeta gráfica NVIDIA	600 €		
6		CUDA Toolkit 6.0	0 €		
			Costes totales		**7.500 €**

Tabla 8. Presupuesto económico del Trabajo de Fin de Grado.

Por lo tanto el coste total de la elaboración del Trabajo de Fin de Grado ha sido de 7.500 €.

8. TRABAJOS FUTUROS

Tras la finalización de la implementación del algoritmo de *matching* para reconocimiento biométrico de iris sobre GPU, existen diversos aspectos sobre los que profundizar relacionados con el presente trabajo. Algunos de ellos se plantean a continuación:

- Aceleración con GPU de los algoritmos de segmentación, normalización y codificación para el reconocimiento biométrico mediante patrón de iris con el fin de aumentar el rendimiento computacional sobre todo el proceso de reconocimiento biométrico de iris.

- Aceleración del algoritmo en *Matlab* sobre GPU: aprovechar las ventajas que presenta el lenguaje *Matlab* en el desarrollo de las aplicaciones de cálculo numérico y programación para una paralelización del algoritmo más sencilla sin necesidad de conocer las arquitecturas de las GPUs y las librerías de cálculo en GPU de bajo nivel.

- Implementación del algoritmo sobre FPGA: Las FPGAs son dispositivos lógicos programables en los que se puede cambiar el conexionado entre los elementos hardware presentes. Tienen la capacidad de realizar operaciones simultáneas para lograr la aceleración hardware de aplicaciones.

- Implementación del algoritmo en OpenGL: OpenGL es una librería estándar para escribir aplicaciones que serán ejecutadas sobre GPUs, permitiendo el cálculo paralelo, similar a CUDA, pero su uso no está restringido a tarjetas gráficas de NVIDIA, sino también a otro tipos de GPUs como AMD.

- Estudio del rendimiento sobre multi-GPU: consiste en el aprovechamiento de múltiples unidades de procesamiento gráfico para mejorar el rendimiento de las aplicaciones drásticamente.

- Cifrado de la base de datos y de los resultados obtenidos tras la comparación de patrones de iris con el fin de aumentar la seguridad y privacidad de los usuarios.

- Desarrollo de una interfaz gráfica de la aplicación basada en algoritmo de *mathing* para una interacción más sencilla con el usuario.

9. REFERENCIAS

[1] Marino Tapiador Mateos, Juan A. Sigüenza Pizarro. *Tecnologías biométricas aplicadas a la seguridad*. Editorial Ra-Ma.

[2] Stan Z.Li, Anil K.Jain. *Encyclopedia of Biometrics*. Editorial Advisor.

[3] Anil K.Jain, Arun Ross, Salil Prabhakar. *An Introduction to Biometric Recognition*. IEEE transactions on circuits and systems for video technology, vol. 14, No. 1.

[4] Anil K.Jain, Arun Ross, Salil Prabhakar. *An Introduction to Biometric Recognition*. IEEE transactions on circuits and systems for video technology, vol. 1, No. 2.

[5] http://www.sistemasbiometricos.cl/web/tag/lectura-de-iris/ (Accedido el 28/07/2014).

[6] http://www.notiseg.com.mx/lector-biometrico-hand-punch.php (Accedido el 28/07/2014).

[7] Rupinder Saini, Narinder Rana. *Comparison of Various Biometric Methods*. International Journal of Advances in Science and Technology. Vol. 2.

[8] http://es.wikipedia.org/wiki/Biometr%C3%ADa (Accedido el 28/07/2014).

[9] http://es.wikipedia.org/wiki/Sistema_de_reconocimiento_facial (Accedido el 30/07/2014).

[10] Pedro Tomé González. *Reconocimiento Automático de Patrones de Iris*. (Proyecto fin de carrera). Universidad Autónoma de Madrid. Junio 2008.

[11] http://www.quirof.cl/anatomia_del_ojo_drjrosasg.htm (Accedido el 05/08/2014).

[12] http://www.monografias.com/trabajos34/ojo-humano/ojo-humano.shtml (Accedido el 05/08/2014).

[13] http://saludbio.com/articulo/constitucion-hidrogenoide-en-iridologia (Accedido el 05/08/2014).

[14] http://www.biometria.sk/en/principles-of-biometrics.html (Accedido el 05/08/2014).

[15] Ashish Kumar Dewangan, Majid Ahmad Siddhiqui. *Human Identification and Verification Using Iris Recognition by Calculating Hamming Distance*. International Journal of Soft Computing and Engineering, vol.2.

[16] http://www.idealtest.org/index.jsp (Accedido el 22/07/2014).

[17] Blaise Barney, Lawrence Livermore National Laboratory. *Introduction to Parallel Computing*. (https://computing.llnl.gov/tutorials/parallel_comp/) (Accedido el 08/08/2014)..

[18] http://en.wikipedia.org/wiki/Parallel_computing (Accedido el 08/08/2014).

[19] Vipin Kumar. Introduction to parallel computing. The Benjamin/Cummings Publishing Company, 1994.

[20]http://webdelprofesor.ula.ve/ingenieria/gilberto/paralela/05_LeyDeAmdahlYMoore.pdf (Accedido el 08/08/2014).

[21] http://lsi.ugr.es/jmantas/pdp/teoria/descargas/PDP_Tema1_Introduccion.pdf (Accedido el 10/08/2014).

[22] Hebert Hoeger. *Introducción a la Computación Paralela*. Centro Nacional de Cálculo Científico Universidad de Los Andes.

[23] A.Arruabarrena, J.Muguerza. *Computadores Paralelos. Computación de Alta Velocidad*. Konputagailuen Arkitektura eta Teknologia saila. Informatika Fakultatea. Euskal Herriko Unibertsitatea.

[24] http://es.wikipedia.org/wiki/CUDA (Accedido el 16/08/2014).

[25] http://www.nvidia.es/object/cuda-parallel-computing-es.html (Accedido el 16/08/2014).

[26] http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accedido el 16/08/2014).

[27] http://3dgep.com/introduction-to-cuda-5-0/ (Accedido el 20/08/2014).

[28]http://bibing.us.es/proyectos/abreproy/11926/fichero/Segunda+parte+TECNOLOGIA+CUDA%252Fi_cuda.pdf (Accedido el 20/08/2014).

[29]http://bibing.us.es/proyectos/abreproy/11926/fichero/Tercera+parte+CODIGO+del+PROYECTO%252FV_OPTIMIZACION+PARA+COMPUTACION+GPU+EN+CUDA.pdf (Accedido el 20/08/2014).

[30] http://www.hpcvl.org/sites/default/files/courseHandouts/CUDA_advanced.pdf (Accedido el 24/08/2014).

[31] http://www.ceta-ciemat.es/attachments/376_udc-ceta-curso-cuda.pdf (Accedido el 24/08/2014).

[32] http://elarmarioinformatico.blogspot.com.es/2010/04/cuda-modelo-de-pgramacion.html (Accedido el 16/08/2014).

[33] http://docs.nvidia.com/cuda/cuda-runtime-api/ (Accedido el 16/08/2014).

[34] Libor Masek. *Recognition of Human Iris Patterns for Biometric Identification*. Bachelor of Engineering degree of the School of Computer Science and Software Engineering, The University of Western Australia, 2003.

[35] Libor Masek, Peter Kovesi. *MATLAB Source Code for a Biometric Identification System Based on Iris Patterns*. The School of Computer Science and Software Engineering, The University of Western Australia. 2003.

GLOSARIO DE ACRÓNIMOS

API: Application Programming Interface

CASIA: Chinese Academy of Sciences Institute of Automation

CUDA: Compute Unified Device Architecture

CPU: Central Processing Unit

EER: Equal Error Rate

FAR: False Acceptance Rate

FER: Failure to Enroll Rate

FPGA: Field Programmable Gate Array

FRR: False Rejection Rate

GPU: Graphics Processing Unit

GPGPU: General Purpose Computing on Graphics Processing Units

HD: Hamming Distance

MIMD: Multiple Instruction, Multiple Data

MISD: Multiple Instruction, Single Data

RAM: Random Access Memory

SIMD: Single Instruction, Multiple Data

SISD: Single Instruction, Single Data